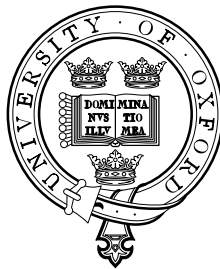


# Lambda Calculus and Types

(complete)

Andrew D. Ker

16 Lectures, Hilary Term 2009



Oxford University Computing Laboratory



# Contents

<b>Introduction To The Lecture Notes</b>	<b>vii</b>
<b>1 Terms, Equational Theory</b>	<b>1</b>
1.1 Inductive Definitions . . . . .	1
1.2 Terms . . . . .	2
1.3 Construction Trees . . . . .	4
1.4 $\lambda$ is a Binding Operator — Alpha-Conversion . . . . .	5
1.5 $\lambda$ Performs Substitution — Beta-Conversion . . . . .	6
1.6 Contexts . . . . .	7
1.7 Formal Theory of the Lambda Calculus . . . . .	8
1.8 Other Lambda Theories . . . . .	10
1.9 A Fixed Point Theorem . . . . .	11
Exercises . . . . .	11
<b>2 Reduction, Consistency</b>	<b>15</b>
2.1 Reduction . . . . .	15
2.2 Beta Reduction . . . . .	17
2.3 Some Special Sets of Terms and Some Special Types of Reduction . . . . .	18
2.4 Properties of Notions of Reduction and Terms . . . . .	19
2.5 Church-Rosser Property of Beta Reduction . . . . .	21
2.6 Consistency . . . . .	23
2.7 Böhm's Theorem . . . . .	25
Exercises . . . . .	26

<b>3</b>	<b>Reduction Strategies</b>	<b>31</b>
3.1	Reduction Strategies . . . . .	31
3.2	Standard Reductions and Standardization . . . . .	34
3.3	More on Head Normal Forms . . . . .	36
	Exercises . . . . .	39
<b>4</b>	<b>Numerals and Recursive Functions</b>	<b>43</b>
4.1	Church Numerals . . . . .	43
4.2	Definability of Total Recursive Functions . . . . .	44
4.3	Undecidability Results . . . . .	47
4.4	Extension to Partial Functions . . . . .	48
	Exercises . . . . .	49
<b>5</b>	<b>Models</b>	<b>53</b>
5.1	Combinatory Logic . . . . .	53
5.2	Simulating Abstraction in Combinatory Logic . . . . .	55
5.3	Translations to and from the Lambda Calculus . . . . .	56
5.4	More Analogies Between CL and $\lambda$ -calculus . . . . .	58
5.5	Combinatory Algebras . . . . .	59
5.6	A Simple Combinatory Algebra . . . . .	61
5.7	$\mathcal{P}\omega$ : A Less Simple Combinatory Algebra . . . . .	62
	Exercises . . . . .	67
<b>6</b>	<b>Simple Types</b>	<b>71</b>
6.1	Simple Type Assignment . . . . .	72
6.2	Lemmas About $TA_\lambda$ . . . . .	75
6.3	The Subject Reduction Theorem . . . . .	77
6.4	Equality and Reduction in $TA_\lambda$ . . . . .	79
6.5	An Alternative Presentation of Simple Types . . . . .	79
	Exercises . . . . .	80
<b>7</b>	<b>Typability and Strong Normalization</b>	<b>83</b>
7.1	Typability . . . . .	83
7.2	Tait's "Reducibility" Proof . . . . .	84
7.3	Consequences of Strong Normalization . . . . .	88
	Exercises . . . . .	89

<b>8</b>	<b>Principal Types</b>	<b>91</b>
8.1	Type Substitutions and Principal Types . . . . .	92
8.2	Lemmas about Type Substitutions . . . . .	94
8.3	Unification . . . . .	97
8.4	The Principal Type Algorithm . . . . .	100
8.5	Consequences of the Principal Type Algorithm . . . . .	106
	Exercises . . . . .	107
<b>A</b>	<b>Answers to Computational Practice Questions</b>	<b>109</b>
<b>B</b>	<b>Sample Finals Questions</b>	<b>119</b>
<b>C</b>	<b>A Primer On Computability</b>	<b>127</b>



# Introduction To The Lecture Notes

## Course

The course is entitled **Lambda Calculus and Types** and it appears in various forms. It is available under: schedule B2 for third year Computer Science or Mathematics and Computer Science undergraduates, course OCS3a in Section B for Mathematics third year undergraduates, schedule I of section A of the MSc in MFoCS, schedule B of the MSc in Computer Science. It is also available to first year D.Phil students at the Computing Laboratory, and anyone else with an interest in the subject.

There will be 16 lectures, and six classes.

## Prerequisites

The course is almost self-contained.

Some knowledge of computability (more precisely, recursive function theory) would be useful for Chapter 4 but it is not necessary: a brief summary of computability basics can be found in Appendix C. (For more in-depth recursive function theory the standard reference is [Cut80], and there are a large number of alternative books on the subject.)

## Syllabus

Terms, formal theories  $\lambda\beta$  and  $\lambda\beta\eta$ , fixed point combinators; reduction, Church-Rosser property of  $\beta$ -reduction and consistency of  $\lambda\beta$ ; reduction strategies, proof that leftmost reduction is normalising; Church numerals, definability of total recursive functions in the  $\lambda$ -calculus, Second Recursion Theorem and undecidability results; combinatory algebras, combinatory completeness, basis; simple types *a la Curry*, type deductions, Subject

Reduction Theorem, strong normalisation and consequences; type substitutions, unification, correctness of Principal Type Algorithm

## Outline of Lectures

Below is an idealised lecture outline. Should time run short, less may be lectured on; undergraduates may be sure that material not covered in lectures will not be examined.

Introductory lecture.

Chapter 1: An introduction to, and motivation for, the language, including definitions of terms, variables, substitution and contexts, conversion. Introduction of the standard theories  $\lambda\beta$  and  $\lambda\beta\eta$ . General  $\lambda$ -theories and properties we might desire of them. Fixed point combinators.  
(about 1.5 lectures)

Chapter 2: A primer on term re-writing, concentrating on notions of reduction over the terms of the  $\lambda$ -calculus.  $\beta$ -reduction, and proof that it is Church-Rosser. Consistency of  $\lambda\beta$ , more discussion of consistency. Böhm's Theorem and inconsistency of equating distinct  $\beta\eta$ -normal forms.  
(2 lectures)

Chapter 3: Reduction strategies. Leftmost and head reduction, head normal forms, Standardisation (using an unproved lemma on advancement of head reduction), leftmost reduction is normalising. More on head normal forms, including Genericity Lemma (unproved) and consistency of equating terms with no hnf. Very briefly: the  $\lambda$ -theories  $\mathcal{H}$  and  $\mathcal{H}^*$ .  
(1.5 lectures)

Chapter 4: Church numerals, some simple arithmetic. Definability of total recursive functions. Gödel numbering. Undecidability results and the Second Recursion Theorem. Briefly: extension to partial functions.  
(2 lectures)

Chapter 5: Definition of combinatory algebras, and some straightforward results about their structure. Combinatory completeness. Translation between combinatory algebras and the  $\lambda$ -calculus;  $\lambda$ -algebras. Presentation of a simple combinatory algebra.



(2 lectures)

Chapter 6: Simple types *a la Curry*. Type contexts and deductions. Typability. Construction lemmas, Subject Reduction Theorem. Lack of type invariance under expansion and equality.

(2 lecture)

Chapter 7: Strong normalisation (via reducibility) and definability consequences. Decidability. Very briefly: PCF.

(1.5 lectures)

Chapter 8: Polymorphism. Type substitution and unification. Principal Type Algorithm (including proof of correctness).

(2.5 lecture)

The interdependence of the chapters, roughly, is according to the diagram in Figure 1.

A word on what we will *not* cover. We will not discuss implementation of the various  $\lambda$ -calculi on real-life machines. This is a large subject in its own right. Three major results of the untyped  $\lambda$ -calculus will be presented, and used, without proof: Böhm's Theorem, a lemma on head reduction, and the Genericity Lemma.

Each chapter of these notes only scratches the surface of a deep and fascinating topic. If you are interested in learning more, there should be suggestions for further reading in the references.

## Reading Material

These notes aim to be a self-contained reference on the basics. For further reading, each chapter begins with a list of suitable material. Unfortunately, there is much inconsistency of notation between different authors, so care must be taken to translate the terminology of each text correctly.

The two most thorough texts are:

- (i) Barendregt's comprehensive survey [Bar84]. The definitive reference on the untyped  $\lambda$ -calculus, but it is an advanced book and only a fraction of it is covered in this course. It does not cover types at all (they are mentioned very briefly in the appendix, and the presentation does not match ours). It appears to be back in print, although expensive. There are many copies in Oxford libraries.

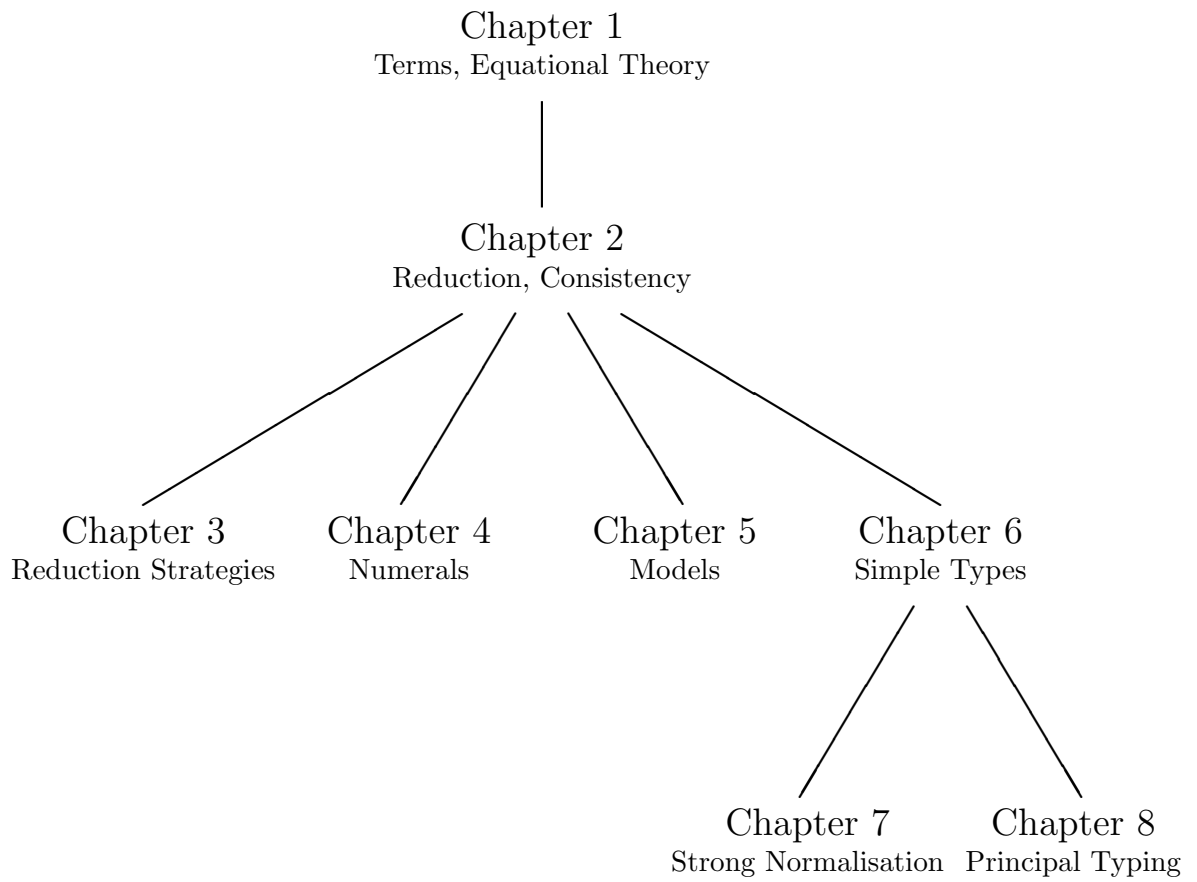


Figure 1: Rough interdependence of the chapters

- (ii) Hindley’s book on simple types [Hin97]. Chapter 1 of this book is a very brief overview of some topics in untyped  $\lambda$ -calculus (too brief to be of much use) but chapters 2 and 3 (with some references to 4 and 5) form the “types” half of this course.

Some other literature to consider reading includes:

- (iii) Hankin’s book [Han94], in some ways ideal because it takes a selection of material of [Bar84] (the selection is quite close to the content of this course) and presents it more slowly. Personally I don’t like the presentation very much, but it may suit you. It is now available very cheaply in paperback and some early parts of the book can be freely downloaded from <http://www.doc.ic.ac.uk/~clh/extracts.pdf>.
- (iv) An excellent set of lecture notes, written by Peter Selinger [Sel07], is available at <http://www.mathstat.dal.ca/~selinger/papers/lambdanotes.pdf>. Its coverage of our topics is more advanced than ours.
- (v) Barendregt and Barendsen’s paper *An Introduction to the Lambda Calculus* [BB94]. A quick and clear overview which includes many of the topics covered in this course. A copy can be downloaded from <http://citeseer.ist.psu.edu/barendregt94introduction.html>. An updated version is at <http://www.cs.chalmers.se/Cs/Research/Logic/TypesSS05/Extra/geuvers.pdf>.
- (vi) Two classic texts on the  $\lambda$ -calculus are [HS86] and [GLT89]. But neither presents the material in the same way as here.
- (vii) Some very nice notes on  $\lambda$ -calculus have been written by Chantal Berline and can be downloaded from <http://www.pps.jussieu.fr/~berline/Cours.html>. Beware that the typed systems she covers are quite different to ours, and that these notes are in French.

## Exercises and Computational Practice

At the end of each chapter are two types of questions:

- (a) Computational Practice, and
- (b) Exercises.

The computational practice questions are simple calculations which do not ask you to construct proofs: they are almost all straightforward. Answers are in the back of the notes. The idea is that you should check your understanding of the basic definitions by trying some of the computational practice questions in your own time.

The exercises have quite variable difficulty and many are substantial. Each week, I will set a selection of exercises to do for the classes. In the classes, I will present answers to the set exercises along with any lessons learned.

Even if your degree course does not require you to hand in work and attend the weekly classes, it would be a good idea to look at the set exercises because some important ideas are only sketched out in lectures and explored properly in the exercises.

Exercises marked  $\star$  are more difficult, slightly off-syllabus, or too much fun to be serious work.

## **Course Website**

Material for this course, including these notes and the list of exercises to do for each class, can be found at <http://web.comlab.ox.ac.uk/teaching/materials08-09/lambda/>.

# Chapter 1

## Terms, Equational Theory

---

**Reading:** [Bar84] 2.1, 6.1; [BB94] 1, 2; [Han94] Ch. 2;  
[Sel07] 2.1–2.3

---

The study of the  $\lambda$ -calculus is of the set of **terms** and **equations** between the terms. Both these concepts (and indeed many others in this course) are defined **inductively**.

We begin by introducing the terms and explaining the role of the symbol  $\lambda$  as a binding operator which performs substitution. This notion is captured by two **equational theories** over terms, and we show the important **First Recursion Theorem** which will later be of vital importance in showing how powerful this very simple language is. We also briefly consider the possibility of other equational theories, and introduce some healthiness conditions for theories to be of interest.

### 1.1 Inductive Definitions

During the course, we define many sets (also relations) using inductive definitions. We will usually use the following notation from proof theory:

A formation rule for a set  $\mathcal{S}$  is written as:

$$\text{(name of the rule)} \quad \frac{P_1 \in \mathcal{S}, \dots, P_n \in \mathcal{S}}{C \in \mathcal{S}} \text{ (side condition)}$$

and means that, if the **premises**  $P_1, \dots, P_n$  are in  $\mathcal{S}$ , and if the **side condition** (which may be a syntactic condition on the premises and/or the conclusion) also holds, then the **conclusion**  $C$  is in  $\mathcal{S}$ . There might be no premises, or no side condition, or neither premises nor side condition. When there are no premises a rule is sometimes called an **axiom**.

The separation of the conditions required to deduce that  $C \in \mathcal{S}$  into the premises (which only talk about membership of  $\mathcal{S}$ ) and the side-condition (which is arbitrary), is traditional.

When we define a set  $\mathcal{S}$  using this style of rules, we mean that  $\mathcal{S}$  is the *least* (with respect to inclusion) set for which all the rules hold. To prove properties of a set  $\mathcal{S}$  defined in such a fashion, we must use induction.

On the other hand, if we already have a set  $\mathcal{S}$  then we can check it against a rule:

**Definition** A rule  $R$  is **valid** for a set  $\mathcal{S}$  if, whenever the side condition and premises hold, so does the conclusion.

(Equivalently, if  $\mathcal{S}$  was defined by a set of rules,  $R$  will be valid if and only if adding it would not change  $\mathcal{S}$ .) To reduce the number of inductive proofs, the following lemma will be useful.

**Lemma 1.1.1** If  $\mathcal{S}$  and  $\mathcal{T}$  are defined inductively using rules, and if every rule for  $\mathcal{S}$  is valid for  $\mathcal{T}$  then  $\mathcal{T} \supseteq \mathcal{S}$ .

## 1.2 Terms

The objects of study in the  $\lambda$ -calculus are the **terms** of the language, and ideas of **convertibility** or **equality** between pairs of these terms. The terms are remarkably simple objects, especially considering that as the course progresses we will show that they can be used as a programming language.

As a starting point, we assume the existence of a countable set of **variables**.

**Definition** A **term** of the untyped  $\lambda$ -calculus is finite string made up of the symbols  $\lambda, (, ), ., ,$  and variable identifiers, the set of terms  $\Lambda$  being defined by the following BNF grammar:

$$s ::= x \mid (\lambda x.s) \mid (s s)$$

where  $s$  is a term and  $x$  a variable.

Written in the form we described earlier for inductive definitions, we would say

$$\frac{}{x \in \Lambda} \quad x \in \mathcal{V} \quad \frac{s \in \Lambda \quad t \in \Lambda}{(st) \in \Lambda} \quad \frac{s \in \Lambda}{(\lambda x.s) \in \Lambda} \quad x \in \mathcal{V}$$

A **subterm** is a subsequence of the symbols of a term which is itself a valid term according to the above grammar. That is, the set of subterms of the term  $s$ , written  $\text{sub}(s)$ , is given inductively by:

$$\begin{aligned}\text{sub}(x) &= \{x\} \\ \text{sub}((\lambda x.s)) &= \{(\lambda x.s)\} \cup \text{sub}(s) \\ \text{sub}((s_1 s_2)) &= \{(s_1 s_2)\} \cup \text{sub}(s_1) \cup \text{sub}(s_2)\end{aligned}$$

A subterm is named a **variable**, an **abstraction**, or an **application**, depending on which clauses of the grammar it matches.

We usually use letters like  $x, y, z, x', x_i$  for variables, and letters like  $s, t, s', s_i$  as meta-variables to range over terms.<sup>1</sup> Do not confuse the formal “object”-variables of the language with meta-variables which we use to reason about terms.<sup>2</sup>

For example, in the term  $(\lambda x.(\lambda y.(x(\lambda z.(yz))))))$ , some of the subterms are  $(yz)$ , an application,  $x$ , a variable, and  $(\lambda y.(x(\lambda z.(yz))))$ , an abstraction.

The symbols  $(, ), .$  are important parts of the language, which disambiguate the structure of expressions. In practice we try to minimise their use, to avoid notational clutter, and we can do so in a safe way with the following conventions:

- (i) Outermost parentheses can be omitted.
- (ii) A sequence of applications associates to the left:  $xyz$  means  $((xy)z)$ .
- (iii) Nested abstractions associate to the right and can be gathered under a single  $\lambda$  sign:  $\lambda xy.xy$  means  $(\lambda x.(\lambda y.(xy)))$ . Unless otherwise bracketed, the body of an abstraction is as much as possible of the rest of the term.
- (iv) When an abstraction occurs on the right of an application the parentheses around it may be omitted:  $x\lambda y.yy$  means  $(x(\lambda y.(yy)))$ .

So the term from the previous example would be written  $\lambda xy.x\lambda z.yz$ . (In practice we often do not use rule (iv), and include the parentheses around the abstraction, because it makes the term easier to read).

To **prove** things about the set of all terms, we usually use induction. There will be one case for variables: prove that the result holds for terms of the form  $x$ . Another case will be for applications: assuming that the result holds for terms  $s$  and  $t$ , prove that it holds for applications  $st$ . The third case will

<sup>1</sup>Unfortunately we can't help using the letter  $u$  sometimes for variables and sometimes for terms. We could instead use the style of [Bar84], [Han94], and many others, where terms are denoted by upper case letters and there is less ambiguity. But years of habit on the part of the lecturer prevents this. Sorry.

<sup>2</sup>Because of  $\alpha$ -conversion, below, we do not need meta-variables to range over the object variables.

be for abstractions: assuming that the result holds for a term  $s$ , prove that it holds for abstractions  $\lambda x.s$ .

This is not the only possible shape for a proof of something for all terms, but it is by far the most common.

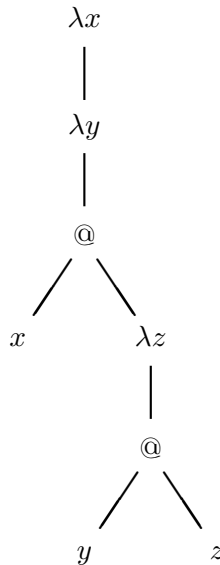
### 1.3 Construction Trees

It is sometimes helpful to think of a term as a tree<sup>3</sup>, the shape of which corresponds to the way the term matches the grammar. There are many (equivalent) definitions of such trees, which go by a number of names including **parse trees** and **term trees**.

**Definition** The **construction tree** of a term  $s$  has nodes labelled  $\lambda x$  (for  $x \in \mathcal{V}$ ) or  $@$ , and leaves labelled by variables. It is given inductively by:

- (i) The construction tree of a variable  $x$  is a single leaf, labelled  $x$ .
- (ii) The construction tree of an abstraction  $\lambda x.s$  consists of a node labelled  $\lambda x$  with a single subtree, which is the construction tree of  $s$ .
- (iii) The construction tree of an application  $st$  consists of a node labelled  $@$  with two subtrees: the left subtree is the construction tree of  $s$  and the right subtree is the construction tree of  $t$ .

For example, the construction tree of the term  $\lambda xy.x\lambda z.yz$  is



We will not draw construction trees very often, but it is very helpful to have the tree structure in mind when doing case analysis or induction on the structure of a term.

<sup>3</sup>Our trees are directed acyclic graphs with an order on the edges leaving each vertex.



## 1.4 $\lambda$ is a Binding Operator — Alpha-Conversion

We introduce the first of two notions of **convertibility** (also called **conversion**) between terms which describe what the symbol  $\lambda$  “does”.

Firstly, it is a binding operator — in the same way that the variable  $x$  in the statement  $\forall x.P$  of formal logic (where the subformula  $P$  possibly contains  $x$ ) is a **dummy variable** which could be swapped for any other fresh variable, so is the variable  $x$  in  $\lambda x.s$ , where the subterm  $s$  possibly contains  $x$ .

**Definition** Each term  $s$  has a set of **free variables**  $FV(s)$ , defined inductively by:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(st) &= FV(s) \cup FV(t) \\ FV(\lambda x.s) &= FV(s) \setminus \{x\} \end{aligned}$$

An **occurrence** of a variable  $x$  in a term  $s$  is said to be **free** if it is in  $FV(s)$ . An occurrence of a variable  $x$  in a term  $s$  is said to be **bound** if it is in a subterm  $t = \lambda x.u$  of  $s$ .

A term  $s$  is said to be **closed** if it has no free variables, i.e.  $FV(s) = \emptyset$ .

Note that a variable can have more than one occurrence, and some of those occurrences might be free while others are bound (an example is  $x$  in  $\lambda y.x\lambda x.x$ )! We will shortly see how this can be avoided. (Incidentally, the  $x$  in  $\lambda x$  does not count as an occurrence; it is a binding operation instead of an occurrence.)

We sometimes refer to a **fresh** variable. A fresh variable is a new name: one which does not occur anywhere in any of the terms we are dealing with.

Since the names of bound variables are not important, we can rename them without changing the meaning of a term. But we should not rename a bound variable to be the same as a free variable:

**Definition** Two terms  $s$  and  $t$  are said to be  **$\alpha$ -convertible**, written  $s \equiv_\alpha t$ , if one can derive the same term from both purely by renaming bound variables to fresh variables.

We consider terms which are  $\alpha$ -convertible to be identical at the syntactic level.<sup>4</sup> We usually omit the subscript  $\alpha$  and write  $s \equiv t$  when  $s$  and  $t$  are syntactically equal in this sense. Later we will define the equality symbol  $=$ , which is weaker. Therefore we will be careful to reserve the  $=$  symbol for the defined equality, and we will use the  $\equiv$  symbol everywhere else.

<sup>4</sup>Formally, we are studying the set  $\Lambda$  of terms quotiented by  $\alpha$ -conversion.

Furthermore, since there is a countably infinite supply of variables, we can make sure that the bound variables of terms do not interfere with each other, or any free variables. We do this by adopting the (slightly vague) **variable convention**:

In any definition, theorem or proof in which only finitely or countably many terms appear, we silently  $\alpha$ -convert them so that bound variables of each term are not the same as the bound variables of any other term, or the free variables of any term.

The set of closed terms,  $\{s \in \Lambda \mid \text{FV}(s) = \emptyset\}$ , is called  $\Lambda^0$ . We talk more often about closed terms than arbitrary ones; closed terms are also sometimes called **combinators** (terminology which has been imported from the parallel subject of **combinatory logic**). During the course we will meet some “standard” combinators which we will generally distinguish from term meta-variables by using boldface Roman letters. We will see much more of combinators in Chapter 5. In the meantime, some standard combinators are given here:

<b>i</b>	$\equiv$	$\lambda x.x$	
<b>b</b>	$\equiv$	$\lambda xyz.x(yz)$	
<b>c</b>	$\equiv$	$\lambda xyz.xzy$	
<b>k</b>	$\equiv$	$\lambda xy.x$	
<b>s</b>	$\equiv$	$\lambda xyz.xz(yz)$	
<b>t</b>	$\equiv$	$\lambda xy.x$	(This is the same combinator as <b>k</b> , but the term
<b>f</b>	$\equiv$	$\lambda xy.y$	is referred to as <b>t</b> in certain situations.)
<b><math>\Omega</math></b>	$\equiv$	$(\lambda x.xx)(\lambda x.xx)$	

## 1.5 $\lambda$ Performs Substitution — Beta-Conversion

Having decided that the  $\lambda$  binds its argument, and having defined one notion of conversion which reflects this, we now give another type of conversion to define what it *does* with the argument.

The term  $\lambda x.s$  behaves like a function, as if  $s$  were parameterised by  $x$ . Application in the  $\lambda$  calculus is like functional application. The power of the untyped  $\lambda$  calculus, and the theorems we will see below, come from the lack of distinction between functions and arguments. A function can even be its own argument (see the fixed point combinator below).

**Definition** The relation on terms  **$\beta$ -convertibility**, which we write infix as  $\beta$ , is defined by the scheme

$$(\lambda x.s)t \beta s[t/x]$$

for any terms  $s$  and  $t$ , where “ $s[t/x]$ ” means “in  $s$  substitute  $t$  for every free occurrence of the variable  $x$ ”.

The moral is that, if  $s(x)$  represents a term in which the variable  $x$  appears (maybe more than once), then  $(\lambda x.s(x))t \beta s(t)$ .

The notion of **substitution** is very important in formal logic. In the  $\lambda$ -calculus substitution is an **implicit** operation. This means that the expression “ $s[t/x]$ ” is not a term (and the symbols  $[, ]$  and  $/$  are not part of the language of the lambda calculus), instead it **denotes** the term that is obtained by substituting  $t$  for free occurrences of  $x$  in  $s$ . Note also that substitution is **unrestricted**, so that any term may be substituted for any variable. This is to be contrasted with some typed languages.

Substitution may be defined recursively as follows:

$$\begin{aligned} x[t/x] &\equiv t, \\ y[t/x] &\equiv y \text{ if } x \neq y, \\ (su)[t/x] &\equiv (s[t/x])(u[t/x]), \\ (\lambda y.s)[t/x] &\equiv \lambda y.(s[t/x]) \text{ assuming } y \neq x \text{ and } y \notin \text{FV}(t), \end{aligned}$$

Note that we can only assume that  $y \notin \text{FV}(t)$ , in the final clause, because of the variable convention. Without this convention, we would need to rename the bound variable  $y$  to avoid the  $\lambda$  capturing and binding a free variable  $y$  in  $t$ . In the absence of the convention, then, we would also need an additional clause:

$$(\lambda y.s)[t/x] \equiv (\lambda z.s[z/y][t/x]) \text{ if } y \equiv x \text{ or } y \in \text{FV}(t)$$

where  $z$  is some variable not occurring in  $s$  or  $t$ .

For example,  $(\lambda xy.yz)[yy/z] \equiv \lambda xu.u(yy)$  (note the need for parentheses here).

## 1.6 Contexts

Substitution has been carefully defined to avoid capture of free variables in the binding operator  $\lambda$ . The following, different, type of substitution is useful for reasoning about syntax. Informally a **context** is a term with “holes” into which other terms can be placed. Formally,

**Definition** The set of **unary contexts** with hole variable  $X$ ,  $\mathcal{C}[X]$ , is the subset of the finite strings of the symbols  $X, \lambda, (, ), .$ , and variable identifiers given by the grammar

$$\mathcal{C}[X] ::= X \mid x \mid (\lambda x.\mathcal{C}[X]) \mid (\mathcal{C}[X]\mathcal{C}[X]).$$

More generally the set of  $n$ -ary contexts  $\mathcal{C}[X_1, \dots, X_n]$  is given by the grammar

$$\mathcal{C}[X_1, \dots, X_n] ::= X_i \mid x \mid (\lambda x. \mathcal{C}[X_1, \dots, X_n]) \mid (\mathcal{C}[X_1, \dots, X_n] \mathcal{C}[X_1, \dots, X_n]).$$

Using the rule-based inductive definition the unary contexts are given by

$$\frac{}{x \in \mathcal{C}[X]} x \in \mathcal{V} \qquad \frac{}{X \in \mathcal{C}[X]}$$

$$\frac{C \in \mathcal{C}[X] \quad D \in \mathcal{C}[X]}{(CD) \in \mathcal{C}[X]} \qquad \frac{C \in \mathcal{C}[X]}{(\lambda x. C) \in \mathcal{C}[X]} x \in \mathcal{V}$$

and similarly for general  $n$ -ary contexts.

We usually use the letters  $C, D$ , etc. to range over contexts and  $X, X_i$  for the contextual variables which are sometimes called **hole variables**.

If  $C$  is context of arity  $n$  we write  $C[t_1, \dots, t_n]$  for the term generated by replacing all occurrences of  $X_i$  in by  $t_i$ . This replacement is regardless of whether this captures free variables of  $t_i$  in abstractions of  $C$ . This “blind” substitution is called **contextual substitution**.

Although the substitution is blind to variable capture, it should not affect the syntactical structure of a term. So the term  $s$  in  $C[s]$  should be considered as a syntactical unit, with disambiguating brackets added around  $s$ , if necessary. For example, if  $C[X] \equiv \lambda x. xyX$  then  $C[xx] \equiv \lambda x. xy(xx)$ , the  $x$  being bound as a result.

Construction trees can be extended to contexts in the obvious way, with leaves labelled either by variables or hole variables. Contextual substitution  $C[s]$  corresponds to grafting the construction tree of  $s$  onto every place where  $X$  occurred in  $C[X]$ .

## 1.7 Formal Theory of the Lambda Calculus

In logic, a **theory** is a set of formulae closed under some notion of provability or derivability. We describe a theory inductively using rules in the style of Section 1.1.

In the formal theories of the  $\lambda$ -calculus, the only formulae are equations between terms — strings of the form “ $s = t$ ” for terms  $s$  and  $t$ . By convention we do not write “ $s = t \in \mathcal{T}$ ” to say that  $s = t$  a member of the set of formulae provably equal in the theory  $\mathcal{T}$ , instead writing  $\mathcal{T} \vdash s = t$  or just  $s = t$  when  $\mathcal{T}$  can safely be deduced from the context.

**Definition** The “standard” theory of the  $\lambda$ -calculus is called  $\lambda\beta$  and is defined by the following rules:

$$\begin{array}{l} \text{(reflexivity)} \quad \frac{}{s = s} \\ \text{(symmetry)} \quad \frac{s = t}{t = s} \\ \text{(transitivity)} \quad \frac{s = t \quad t = u}{s = u} \end{array}$$

(These ensure that  $=$  is an equivalence relation.)

$$\begin{array}{l} \text{(application)} \quad \frac{s = s' \quad t = t'}{st = s't'} \\ \text{(abstraction)} \quad \frac{s = t}{\lambda x.s = \lambda x.t} \end{array}$$

(These ensure that  $=$  is a **congruence**, meaning that if  $s = t$  then  $C[s] = C[t]$  for all contexts  $C[X]$ .)

$$(\beta) \quad \frac{}{(\lambda x.s)t = s[t/x]}$$

(This says that  $\beta$ -convertible terms are equal.)

A formal proof of an equation in  $\lambda\beta$  is therefore a “proof tree” constructed from these rules. For example,

$$\frac{\frac{\frac{}{(\lambda xy.x)p = \lambda y.p}(\beta) \quad \frac{}{q = q}(\text{refl})}{(\lambda xy.x)pq = (\lambda y.p)q}(\text{app}) \quad \frac{}{(\lambda y.p)q = p}(\beta)}{(\lambda xy.x)pq = p}(\text{trans})}{p = (\lambda xy.x)pq}(\text{sym})$$

is a proof that  $\lambda\beta \vdash p = (\lambda xy.x)pq$ . But in practice we will not want to draw proof trees, instead silently using the fact that  $=$  is an equivalence relation and a congruence, and only noting the uses of the  $\beta$  rule, so we should only need to write a linearised proof like  $\lambda\beta \vdash (\lambda xy.x)pq = (\lambda y.p)q = p$ .

Another “standard” theory is called  $\lambda\beta\eta$  and is defined by all the rules of  $\lambda\beta$  plus:

$$(\eta) \quad \frac{}{\lambda x.sx = s} \quad x \notin \text{FV}(s)$$

The rule  $\eta$  of the theory  $\lambda\beta\eta$  is also motivated by the principle that all terms are functions — since  $\lambda\beta \vdash (\lambda x.sx)y = sy$  then  $\lambda x.sx$  and  $s$  are equated when applied to any “argument”. We define  $\eta$ -**conversion** to be conversion

between terms of these forms. If we wish to take an **extensional** viewpoint then the  $\eta$ -convertible terms should be equated.

However it is generally the case that programming languages based on the  $\lambda$ -calculus do not include  $\eta$ -conversion.

The theory  $\lambda\beta$  is sometimes called “the  $\lambda$ -calculus”. But, although much of the study of the  $\lambda$ -calculus is concerned with the study of this theory, there are many other theories to consider.  $\lambda\beta\eta$  is sometimes called “the extensional  $\lambda$ -calculus”.

## 1.8 Other Lambda Theories

There are theories of interest besides just  $\lambda\beta$  and  $\lambda\beta\eta$ , but they wouldn't be anything to do with the  $\lambda$ -calculus unless they at least equate all  $\beta$ -convertible terms, and are congruences:

**Definition** If  $\mathcal{T}$  is a set of equations between terms then we call it a  **$\lambda$ -theory** if all the rules of  $\lambda\beta$  are valid for  $\mathcal{T}$ .

Note that this is stronger than simply requiring that  $\lambda\beta \subseteq \mathcal{T}$ . Traditionally, we only consider sets of closed equations, i.e. equations between closed terms. This turns out to be a technical nicety which makes no difference in practice — for an explanation see [Bar84, §4.1]. A useful definition for constructing  $\lambda$ -theories is the following:

**Definition** If  $\mathcal{T}$  is a set of equations between terms then we write  $\lambda\beta + \mathcal{T}$  for the (least) theory which contains all the equations in  $\mathcal{T}$  and is closed under the formation rules of  $\lambda\beta$ .

(This is equivalent to adding all the equations of  $\mathcal{T}$  to the system  $\lambda\beta$ , as additional axioms.)

It is easy to see that a set of equations between terms  $\mathcal{T}$  is a  $\lambda$ -theory if  $\lambda\beta + \mathcal{T} = \mathcal{T}$ . Clearly,  $\lambda\beta$  and  $\lambda\beta\eta$  are  $\lambda$ -theories, and one can check that  $\lambda\beta\eta = \lambda\beta + \{\lambda xy.xy = \lambda x.x\}$ . We shall see some examples of other  $\lambda$ -theories later.

There are some important questions we should ask about a  $\lambda$ -theory  $\mathcal{T}$  (and we should certainly ask them of  $\lambda\beta$  and  $\lambda\beta\eta$ ):

- (i) Is  $\mathcal{T}$  **consistent**? In the  $\lambda$ -calculus, we say that a  $\lambda$ -theory is consistent if it does *not* equate all terms<sup>5</sup>. For  $\lambda\beta$  and  $\lambda\beta\eta$  the answer is yes — see Theorem 2.6.1.

---

<sup>5</sup>Indeed, some authors require consistency for  $\mathcal{T}$  even to be called a  $\lambda$ -theory.

- (ii) Is equality in  $\mathcal{T}$  decidable? For  $\lambda\beta$  and  $\lambda\beta\eta$  the answer is no — see Section 4.3 and Exercise 4.10 in particular.
- (iii) Is  $\mathcal{T}$  **Hilbert-Post complete** (also called **maximally consistent**)? That is, is every theory which strictly contains  $\mathcal{T}$  inconsistent? For  $\lambda\beta$  the answer is clearly no because  $\lambda\beta\eta$  is a consistent extension of  $\lambda\beta$ . For  $\lambda\beta\eta$  the answer is also no — see Section 3.3 and Lemma 3.3.4. Also see Exercise 3.13 for an example of a Hilbert-Post complete  $\lambda$ -theory.
- (iv) (More vaguely) Does  $\mathcal{T}$  have an operational interpretation, i.e. some meaning in the context of programming?

## 1.9 A Fixed Point Theorem

The ability to apply a term to itself leads to some remarkable properties of the  $\lambda$ -calculus. One particularly important such result is given here.

**Definition** For terms  $f$  and  $u$ ,  $u$  is said to be a **fixed point** of  $f$  (in  $\lambda\beta$ ) if  $\lambda\beta \vdash fu = u$ .

Since  $\lambda\beta \subseteq \mathcal{T}$  for any  $\lambda$ -theory  $\mathcal{T}$ , a fixed point of  $f$  in  $\lambda\beta$  is a fixed point of  $f$  in any  $\lambda$ -theory. In the  $\lambda$ -calculus, every term has a fixed point:

**Theorem 1.9.1 (First Recursion Theorem)** Let  $f$  be a term. Then there is a term  $u$  which is a fixed point of  $f$ .

**Proof** In fact the fixed points can be computed in the  $\lambda$ -calculus itself, using a **fixed point combinator**. A fixed point combinator is a closed term  $s$  such that for all terms  $f$ ,  $f(sf) = sf$ . In other words,  $sf$  is a fixed point of  $f$ , for every  $f$ .

There are many fixed point combinators in the theory. Two well-known examples are:

- (i) Curry’s “paradoxical” combinator:  $\mathbf{y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
- (ii) Turing’s fixed point combinator:  $\mathbf{\theta} \equiv (\lambda xy.y(xxy))(\lambda xy.y(xxy))$

It remains to prove that these terms really are fixed point combinators, which we will do in the lectures. ■

## Computational Practice

**1.a** Which of the following are either terms or terms with parentheses unambiguously removed?

- (i)  $\lambda x.xxx.$
- (ii)  $\lambda\lambda x.x.$
- (iii)  $\lambda y.(\lambda x.x).$
- (iv)  $\lambda xy.(xy)(\lambda z.z).$
- (v)  $\lambda uv.((xy)xy)y).$

**1.b** Write these terms with the minimum necessary parentheses:

- (i)  $(\lambda x.(\lambda y.(\lambda z.(z(xy))))).$
- (ii)  $(\lambda y.(((yy)y)y)).$
- (iii)  $(\lambda y.(yy)((\lambda x.x)(\lambda x.x))).$

**1.c** For each occurrence of a variable in each of these terms, say whether it is free or bound:

- (i)  $\lambda xyz.xyz.$
- (ii)  $\lambda xyz.yz(\lambda p.xyz).$
- (iii)  $\lambda xy.yz(\lambda z.zz).$

**1.d** Perform the following substitutions, renaming bound variables where necessary in order to follow the variable convention:

- (i)  $(xyz)[y/z].$
- (ii)  $(\lambda x.x)[y/z].$
- (iii)  $(\lambda y.xy)[zz/x].$
- (iv)  $(\lambda y.xy)[yy/x].$
- (v)  $C[z]$ , where  $C[X] \equiv \lambda z.Xz.$
- (vi)  $C[zy]$ , where  $C[X] \equiv \lambda xy.yXy.$

**1.e** Draw “proof trees” in  $\lambda\beta$  of the following equalities:

- (i)  $(\lambda x.x)(\lambda y.yy) = \lambda y.yy.$
- (ii)  $\lambda pq.q = \lambda p.(\lambda x.x)(\lambda q.q).$
- (iii)  $(\lambda xy.y)pq = q.$
- (iv)  $\lambda pq.(\lambda xy.y)pq = \lambda ab.b.$

## Exercises

### 1.1

- (i) Rewrite  $((xy)(\lambda y.(\lambda z.(z(xy)))))$  using the minimum number of parentheses.



(ii) Write the term  $(\lambda xyz.xy(xz))\lambda xy.x$  in full syntax.

**1.2** Draw the construction tree of the combinator  $\mathbf{y}$ , and list all its sub-terms.

**1.3** List all the free variables in

(i)  $\lambda xy.(\lambda u.uvxy)z$

(ii)  $\lambda xy.z(\lambda u.uvxy)$

(iii)  $\lambda wx.z(\lambda u.uvwx)$

(iv)  $\lambda vw.z(\lambda z.uvww)$

(v)  $\lambda yx.z(\lambda u.uwyy)$

Which of these five terms are identified by  $\alpha$ -conversion (i.e. which are actually the same as each other)?

**1.4** Perform the following substitutions:

(i)  $(\lambda x.yx)[yz/x]$

(ii)  $(\lambda y.xy)[yx/x]$

(iii)  $(\lambda z.(\lambda x.yx)xz)[zx/x]$

(iv)  $C[yz]$  where  $C[X] \equiv \lambda z.(\lambda x.yx)Xz$

**1.5** [Nested substitution]. Show that, for any variable  $x$  distinct from  $y$ , any terms  $s$  and  $t$ , and term  $u$  not containing  $x$  as a free variable,

$$s[t/x][u/y] \equiv s[u/y][t[u/y]/x].$$

You will need to use induction on the structure of the term  $s$ .

**1.6** Prove the following in  $\lambda\beta$ :

(i) if  $s = t$  then  $s[u/x] = t[u/x]$  for any term  $u$ .

(ii) if  $s = t$  then  $u[s/x] = u[t/x]$  for any term  $u$ .

(iii) if  $s = t$  and  $p = q$  then  $s[p/x] = t[q/x]$ .

Your proofs should look like trees, using axioms and rules of  $\lambda\beta$  to deduce the equations. (The same proofs show that these equations also hold in  $\lambda\beta\eta$ .)

**1.7** Prove that if  $s = t$  then for any context  $C[X]$ ,  $C[s] = C[t]$ . (Again, this holds in both  $\lambda\beta$  and  $\lambda\beta\eta$ .) You will need to use induction on the structure of  $C[X]$ .

**1.8**

(i) Find a term  $s$  such that, for all terms  $t$  and  $u$ ,  $stu = ut$  (in  $\lambda\beta$ ).

(ii) Show that exists a term  $s$  such that, for all terms  $t$ ,  $st = ss$  (in  $\lambda\beta$ ).

[Hint: do not use the First Recursion Theorem. Use common sense. What does  $s$  do with its “argument(s)” in each case?]

**1.9** Construct:

(i) for each closed term  $s$ , a closed term  $t_1$  such that  $t_1 = t_1s$ .

(ii) for each closed term  $s$ , a closed term  $t_2$  such that  $t_2(\lambda x.x)ss = t_2s$ .

[Hint: to solve equations of the shape  $\dots u \dots = \dots u \dots$ , try to find a sufficient condition of the form  $u = (\dots)u$  and then use the First Recursion Theorem.]

**1.10** Show that there is *no* term  $f$  such that for all terms  $s$  and  $t$ ,  $f(st) = s$ .

[Hint: Use the First Recursion Theorem. You may assume that  $\lambda\beta$  is consistent.]

**1.11** Show that every fixed point combinator can be characterised as a fixed point of a term  $G$ . Find  $G$ .

**\*1.12** Show that  $\mathbf{y}, \mathbf{y}G, \mathbf{y}GG, \dots$  are all fixed point combinators, not equated in  $\lambda\beta$ .

[You will need to use a result from Chapter 2.]

**\*1.13** Show that the axiom scheme  $\lambda x.sx = s$ , for  $x \notin \text{FV}(s)$ , cannot be derived in the system  $\lambda\beta$ .

[You will need to use a result from Chapter 2.]

**1.14** Show that, for any term  $s$  which can be proved equal to an abstraction in  $\lambda\beta$ ,  $\lambda\beta \vdash \lambda x.sx = s$  for  $x$  not occurring free in  $s$ . Note carefully the relationship with the above exercise.

**\*1.15** Write down a short term (less than 50 symbols say), in which the free variable  $z$  is only allowed to occur once, but which is provably equal in  $\lambda\beta$  to a term containing the same variable  $z$  over 1000 times.

**\*1.16** [A modification of a result by Klop]. Define:

$\heartsuit \equiv \lambda abcdefghijklmnopqrstuvw.w(\text{elovethelecturesbyandrew})$ , and

$\star \equiv \heartsuit$ .

Show that  $\star$  is a fixed point combinator. Make up your own amusing variations on this question.

## Chapter 2

# Reduction, Consistency

---

**Reading:** [Bar84] 3.1, 3.2, 3.3, parts of 10.4; [BB94] 4;  
[Han94] Ch. 3 esp. 3.1–3.3, 2.5; [Sel07] 2.4, 2.5, 4

---

Intuitively, substitution is an asymmetric relation. Although in the theory  $\lambda\beta$  both  $(\lambda x.s)t = s[t/x]$  and  $s[t/x] = (\lambda x.s)t$ , we might want to concentrate on the “reduction” half, namely “ $(\lambda x.s)t \rightarrow s[t/x]$ ”. In particular, were we to *implement* the  $\lambda$ -calculus as a programming language, we would expect computation to proceed by reductions.

We introduce the idea of  **$\beta$ -reduction**, which is related to the equational theory  $\lambda\beta$ .  $\beta$ -reductions are not, in general, unique but analysis shows that  $\beta$ -reduction has a property called **Church-Rosser**. This prevents any contradictory reduction paths.

Importantly, the study of  $\beta$ -reduction allows us to prove that the equational theory  $\lambda\beta$  is consistent. We end the chapter with a wider examination of consistency, including the statement of the important syntactic result known as **Böhm’s Theorem**.

The subject of reduction in the  $\lambda$ -calculus is part of a very wide subject called **term rewriting**. Many of the definitions given here can be made more general, but we choose to focus on their application only to the  $\lambda$ -calculus.

### 2.1 Reduction

Reduction, as an abstract concept, can take place on any set. Here we only consider the set of terms  $\Lambda$ .

**Definition** A **notion of reduction** over  $\Lambda$ , also called a **redex rule**, is any binary relation on  $\Lambda$ .

A notion of reduction may be of more interest if reduction can also be carried out “inside” a term, we also sometimes want reflexive, transitive, or (less commonly) symmetric versions. Some notions of reduction will have these properties by definition, others we can give these properties by adding some rules.

**Definition**

- (i) For any notion of reduction  $R$ , **one-step  $R$ -reduction** is the **compatible closure** of  $R$ , the relation (on terms)  $\rightarrow_R$  defined by the following rules.

$$\begin{array}{l}
 \text{(R)} \quad \frac{}{s \rightarrow_R t} \quad (\langle s, t \rangle \in R) \\
 \text{(reduction on the left)} \quad \frac{s \rightarrow_R t}{su \rightarrow_R tu} \\
 \text{(reduction on the right)} \quad \frac{s \rightarrow_R t}{us \rightarrow_R ut} \\
 \text{(reduction under abstraction)} \quad \frac{s \rightarrow_R t}{\lambda x.s \rightarrow_R \lambda x.t}
 \end{array}$$

That is,  $\rightarrow_R$  is the least relation containing  $R$  which is **compatible** with the formation rules of the  $\lambda$ -calculus. We pronounce  $s \rightarrow_R t$  as “ $s$   $R$ -reduces to  $t$  in one step”.  $t$  is called the **reduct** of  $s$ .

- (ii) The relation  $\rightarrow_{\bar{R}}$  is the reflexive closure of  $\rightarrow_R$ . This is equivalent to adding a reflexivity rule to the above four.
- (iii) The relation  $\rightarrow_R^+$  is the transitive closure of  $\rightarrow_R$ . This is equivalent to adding a transitivity rule.
- (iv) The relation  $\twoheadrightarrow_R$  is the reflexive, transitive closure of  $\rightarrow_R$ . This is equivalent to adding both reflexivity and transitivity rules. We pronounce  $s \twoheadrightarrow_R t$  as “ $s$   $R$ -reduces to  $t$ ”.
- (v) The relation  $=_R$  is the reflexive, symmetric, transitive closure of  $\rightarrow_R$ . This is equivalent to adding reflexivity, symmetry, and transitivity rules. We pronounce  $s =_R t$  as “ $s$  is  $R$ -convertible to  $t$ ”.

It is also helpful to find other characterisations of  $\twoheadrightarrow_R$  and  $=_R$ , decomposing them into sequences of  $\rightarrow_R$ , to reduce the complexity of some later proofs.

**Lemma 2.1.1** Let  $R$  be a notion of reduction over  $\Lambda$ .

- (i)  $s \rightarrow_R^+ t$  if and only if for some  $n \geq 1$  and terms  $s_1, \dots, s_n$ ,

$$s \equiv s_0 \rightarrow_R s_1 \rightarrow_R \cdots \rightarrow_R s_n \equiv t$$

- (ii)  $s \twoheadrightarrow_R t$  if and only if for some  $n \geq 0$  and terms  $s_1, \dots, s_n$ ,

$$s \equiv s_0 \rightarrow_R s_1 \rightarrow_R \cdots \rightarrow_R s_n \equiv t$$

- (iii)  $s =_R t$  if and only if for some  $n \geq 0$  and terms  $s_0, \dots, s_{n-1}$  and  $t_1, \dots, t_n$ ,

$$s \equiv s_0 \begin{array}{c} \swarrow \\ t_1 \\ \searrow \end{array} s_1 \begin{array}{c} \swarrow \\ t_2 \\ \searrow \end{array} \dots \begin{array}{c} \swarrow \\ t_{n-1} \\ \searrow \end{array} s_{n-1} \begin{array}{c} \swarrow \\ t_n \equiv t \\ \searrow \end{array}$$

**Proof** All by induction. ■

Also, we can unpick the effect of the compatible closure rules as follows. A **one-holed** unary context  $C[X]$  is a context where the only hole variable  $X$  occurs precisely once.

**Lemma 2.1.2**  $s \rightarrow_R t$  if and only if there is some one-holed context unary  $C[X]$  and terms  $u, v$  such that  $s \equiv C[u]$ ,  $t \equiv C[v]$ , and  $\langle u, v \rangle \in R$ .

**Proof** Exercise. ■

The term  $u$  is called a  **$R$ -redex**, and we say that it is **contracted** to  $v$ .

## 2.2 Beta Reduction

The notion of reduction in which we are most interested is that which embodies  $\beta$ -conversion. Recall from the previous chapter,

**Definition** The notion of reduction called  **$\beta$ -reduction** is the relation  $\beta$  which is the set

$$\beta = \{ \langle (\lambda x.s)t, s[t/x] \rangle \mid s, t \in \Lambda \}$$

The one-step  $\beta$ -reduction,  $\rightarrow_\beta$ , is the compatible closure of this relation, with  $\twoheadrightarrow_\beta$  and  $=_\beta$  also defined in the usual way.

Some examples: the redexes of  $\beta$ -reduction are (sub)terms of the form  $(\lambda x.p)q$ . For any term  $s$ ,  $\mathbf{is} \rightarrow_\beta s$ . Also, let  $\mathbf{\Omega}$  be the standard combinator  $(\lambda x.xx)(\lambda x.xx)$ , then  $\mathbf{\Omega} \rightarrow_\beta \mathbf{\Omega}$  and in fact  $\mathbf{\Omega}$  is the only term  $t$  such that  $\mathbf{\Omega} \twoheadrightarrow_\beta t$  (exercise). (The set of all terms  $t$  such that  $s \twoheadrightarrow_\beta t$  are called the **reducts** of  $s$ .)

Recall the fixed point combinators  $\mathbf{y}$  and  $\mathbf{\theta}$ . The reader may want to check that  $\mathbf{\theta}f \rightarrow_\beta f(\mathbf{\theta}f)$  and  $\mathbf{y}f =_\beta f(\mathbf{y}f)$  but it is not necessarily the case that  $\mathbf{y}f \twoheadrightarrow_\beta f(\mathbf{y}f)$ .

The congruence and equivalence relation  $=_\beta$  on  $\Lambda$  is sometimes called  **$\beta$ -conversion** but that name is also sometimes used for the relation  $\beta$ , so beware. In fact, the relation  $=_\beta$  turns out to be the same as the equality of the formal theory  $\lambda\beta$ .

**Theorem 2.2.1** For terms  $s$  and  $t$ ,  $s =_{\beta} t$  if and only if  $\lambda\beta \vdash s = t$ . We say that  $\beta$ -reduction **implements**  $\lambda\beta$ .

**Proof** Exercise. ■

Because of this connection between  $\rightarrow_{\beta}$  and equality in  $\lambda\beta$  we will be able to use the reduction relation to help with analysis of  $\lambda\beta$  – in particular proving that the theory is consistent.

We can also introduce a notion of reduction to capture the theory  $\lambda\beta\eta$ :

**Definition**  $\eta^{\text{red}}$  is the relation  $\{\langle \lambda x.sx, s \rangle \mid s \in \Lambda, x \notin \text{FV}(s)\}$  and  $\eta^{\text{exp}}$  the converse relation  $\{\langle s, \lambda x.sx \rangle \mid s \in \Lambda, x \notin \text{FV}(s)\}$ . We call these  $\eta$ -**reduction** and  $\eta$ -**expansion** respectively.

We write  $\beta\eta$  for  $\beta \cup \eta^{\text{red}}$ . In Exercise 2.16 it is shown that the notion of reduction  $\beta\eta$  implements  $\lambda\beta\eta$ .

## 2.3 Some Special Sets of Terms and Some Special Types of Reduction

In analysis of  $\beta$ -reduction it is sometimes helpful to note the effects which one-step reduction can have:

**Definition** Suppose that  $s \rightarrow_{\beta} t$ , in which a redex  $(\lambda x.p)q$  in  $s$  is contracted to  $p[q/x]$  in  $t$ .

- (i) We say that the reduction is **cancelling** if  $x$  does not occur free in  $p$  (hence the subterm  $q$  is deleted in  $t$ ).
- (ii) We say that the reduction is **duplicating** if  $x$  occurs free at least twice in  $p$  (hence the subterm  $q$  is replicated in  $t$ ).

Parallel to these special types of reduction are some special types of terms (not covered in detail in this course, but quite interesting in their own right):

**Definition**

- (i) A term  $s$  is called a  **$\lambda I$ -term** if, for each abstraction  $\lambda x.p$  occurring in  $s$ ,  $x$  occurs free in  $p$  at least once.
- (ii) A term  $s$  is called an **affine** term if, for each abstraction  $\lambda x.p$  occurring in  $s$ ,  $x$  occurs free in  $p$  at most once, and each free variable in  $s$  occurs exactly once.
- (iii) A term  $s$  is called a **linear** term if, for each abstraction  $\lambda x.p$  occurring in  $s$ ,  $x$  occurs free in  $p$  exactly once, and each free variable in  $s$  occurs exactly once.

There are some other names for these sets: linear terms are sometimes called **unitary** or *BCI* terms; affine terms are sometimes called *BCK* terms. All three sets of terms are well-behaved with respect to  $\beta$ -reduction (also  $\beta\eta$ -reduction), i.e.

**Lemma 2.3.1** If  $s$  is a  $\lambda I$ -term (respectively, affine term, linear term) and  $s \rightarrow_{\beta} t$  then  $t$  is also a  $\lambda I$ -term (affine term, linear term).

**Proof** Easy and omitted. ■

The special sets of terms connect with the special types of reduction as follows:

**Lemma 2.3.2** Every  $\beta$ -reduction of a  $\lambda I$ -term is non-cancelling; every  $\beta$ -reduction of an affine term is non-duplicating, and every  $\beta$ -reduction of a linear term is non-cancelling and non-duplicating.

**Proof** Exercise. ■

The  $\lambda I$ -terms are particularly well studied because Church's work focussed on them; they have quite nice behaviour while retaining strong computational power. See [Bar84, Chapter 9] for some of their nice properties.

## 2.4 Properties of Notions of Reduction and Terms

All of the following are definitions from the general theory of term rewriting, although we only consider their application to notions of reduction over  $\Lambda$ .

**Definition** Let  $\rightarrow_R$  be a binary relation and  $\rightarrow_R^*$  its reflexive transitive closure (usually,  $\rightarrow_R^*$  will be the compatible closure of a notion of reduction over  $\Lambda$ , but the definitions work for any binary relation.)

- (i) A term  $s$  is **in  $R$ -normal form** (often abbreviated to **in  $R$ -NF**, or **in normal form** if  $R$  is obvious) if there is no term  $t$  such that  $s \rightarrow_R t$ .
- (ii) A term  $s$  **has a  $R$ -normal form** if there is a term  $t$  in normal form with  $s \rightarrow_R^* t$ . Equivalent terminology is to say that  $s$  is **normalizable (w.r.t.  $\rightarrow_R$ )**. Then we say that  $t$  is a normal form for  $s$ .
- (iii) A term  $s$  is **strongly normalizable** (w.r.t.  $\rightarrow_R$ ) if there does not exist an infinite sequence  $t_1, t_2, \dots$  such that  $s \rightarrow_R t_1 \rightarrow_R t_2 \rightarrow_R \dots$ . That is, every sequence of one-step  $R$ -reductions starting with  $s$  must be finite and end with a normal form.
- (iv)  $R$  is **weakly normalizing** if every term is normalizable w.r.t.  $R$ .
- (v)  $R$  is **strongly normalizing** if every term is strongly normalizable w.r.t.  $R$ .

We give some examples. The term  $\lambda x.x$  is in  $\beta$ -normal form, and the term  $(\lambda x.x)(\lambda x.x)$   $\beta$ -reduces to  $\lambda x.x$  so has a  $\beta$ -normal form. This term is also strongly normalizing.

The term  $\Omega$  is not even weakly normalizable, because it reduces to itself and nothing else. Thus the term  $(\lambda xy.y)\Omega\mathbf{i}$  is weakly normalizable — a finite sequence of reductions is  $(\lambda xy.y)\Omega\mathbf{i} \rightarrow_{\beta} (\lambda y.y)\mathbf{i} \rightarrow_{\beta} \mathbf{i}$  — but not strongly normalizable because it also reduces to itself.

Because of “looping” terms,  $\beta$ -reduction is not even weakly normalizing. In practice it is not a “problem” for a notion of reduction to be not normalizing, as long as it has the Church-Rosser property defined below.

For a notion of reduction where there are terms which are weakly but not strongly normalizable, it becomes important to choose the “right” reductions to make if one ever expects to arrive at a normal form. This leads to the idea of a **reduction strategy** — a subset of the one-step reduction which is a function. That is, each term has at most one reduct. Reduction strategies, especially those for  $\beta$ -reduction, are an important part of the study of the  $\lambda$ -calculus and for its implementation as a programming language, and we study them in the next chapter.

Here are some desirable properties of notions of reduction:

**Definition** Let  $\rightarrow_R$  be a binary relation<sup>1</sup> and  $\twoheadrightarrow_R$  its reflexive transitive closure.

- (i)  $\rightarrow_R$  satisfies the **diamond property** if  $s \rightarrow_R t_1$  and  $s \rightarrow_R t_2$  implies  $t_1 \rightarrow_R u$  and  $t_2 \rightarrow_R u$  for some  $u$ .
- (ii)  $\rightarrow_R$  is **Church-Rosser** if  $\twoheadrightarrow_R$  satisfies the diamond property. That is, if  $s \twoheadrightarrow_R t_1$  and  $s \twoheadrightarrow_R t_2$  implies  $t_1 \twoheadrightarrow_R u$  and  $t_2 \twoheadrightarrow_R u$  for some  $u$ .
- (iii)  $\rightarrow_R$  has the **unique normal form** property if  $s \twoheadrightarrow_R t_1$  and  $s \twoheadrightarrow_R t_2$ , for normal forms  $t_1$  and  $t_2$ , implies that  $t_1 \equiv t_2$ .

These properties are related:

**Lemma 2.4.1**

- (i) If  $\rightarrow_R$  satisfies the diamond property then  $\twoheadrightarrow_R$  satisfies the diamond property.
- (ii) If  $\rightarrow_R$  is Church-Rosser then  $s =_R t$  if and only if there is some  $u$  with  $s \twoheadrightarrow_R u$  and  $t \twoheadrightarrow_R u$ .
- (iii) If  $\rightarrow_R$  is Church-Rosser then it has the unique normal form property.

**Proof** (i) and (ii) are by “diagram chase”.

<sup>1</sup>usually the compatible closure of a notion of reduction  $R$ , although these definitions make sense for any binary relation.



For (iii), suppose  $s \rightarrow_R t_1$  and  $s \rightarrow_R t_2$ . Since  $R$  is Church-Rosser, there is a term  $u$  with  $t_1 \rightarrow_R u$  and  $t_2 \rightarrow_R u$ . If  $t_1$  and  $t_2$  are normal forms, then  $u \equiv t_1$  and  $u \equiv t_2$ , hence  $t_1 \equiv t_2$ . ■

Note that  $\beta$ -reduction does not satisfy the diamond property, because of cancelling and duplicating reductions.

## 2.5 Church-Rosser Property of Beta Reduction

This section is devoted to the proof that  $\beta$ -reduction has the Church-Rosser property. Consequences of this are presented in the following section.

We prove the Church-Rosser property in an indirect manner, via an auxiliary notion of **parallel  $\beta$ -reduction**. This proof is due to Martin-Löf and Tait, and is relatively easy to construct. A direct proof, which does not require this auxiliary notion and is more enlightening for advanced readers, can be found in [Bar84, §11.1].

**Definition** The binary relation over terms called **parallel  $\beta$ -reduction**, written  $\rightarrow_{\parallel}$ , is defined by the rules:

$$\begin{array}{ll} \text{(refl)} \frac{}{s \rightarrow_{\parallel} s} & \text{(\parallel-app)} \frac{s \rightarrow_{\parallel} s' \quad t \rightarrow_{\parallel} t'}{st \rightarrow_{\parallel} s't'} \\ \text{(abs)} \frac{s \rightarrow_{\parallel} s'}{\lambda x.s \rightarrow_{\parallel} \lambda x.s'} & \text{(\parallel-\beta)} \frac{s \rightarrow_{\parallel} s' \quad t \rightarrow_{\parallel} t'}{(\lambda x.s)t \rightarrow_{\parallel} s'[t'/x]} \end{array}$$

Obviously this is closely related to  $\beta$ -reduction, but one step of parallel  $\beta$ -reduction can include a lot more computation than one step of  $\beta$ -reduction.

We will show that  $\rightarrow_{\parallel}$  satisfies the diamond property, and that  $\rightarrow_{\beta}$  is the transitive closure of  $\rightarrow_{\parallel}$ , then by Lemma 2.4.1  $\rightarrow_{\beta}$  satisfies the diamond property, hence  $\beta$ -reduction is Church-Rosser. Note that we have introduced  $\rightarrow_{\parallel}$ , the one-step reduction, directly. It is already a congruence, so we did not need to take compatible closure of some other relation. Unlike  $\rightarrow_{\beta}$ , there isn't a succinct notion of reduction of which  $\rightarrow_{\parallel}$  is the compatible closure.

We proceed as follows:

**Lemma 2.5.1 (Substitution for  $\parallel$ - $\beta$ )** If  $s \rightarrow_{\parallel} s'$  and  $t \rightarrow_{\parallel} t'$  then  $s[t/x] \rightarrow_{\parallel} s'[t'/x]$ .

**Proof** The result is proved for all terms  $s'$ ,  $t$ , and  $t'$  simultaneously, by

induction on the structure of  $s$ .

So suppose the result holds for all subterms of  $s$ , and  $s \rightarrow_{\parallel} s'$ . We perform case analysis on the rule showing that  $s \rightarrow_{\parallel} s'$ :

- (refl) Then  $s \equiv s'$ . We examine the structure of  $s$  by case analysis:

$s$	$s[t/x]$	$s'[t'/x] \equiv s[t'/x]$
$x$	$t$	$t'$
$y$	$y$	$y$
$pq$	$p[t/x]q[t/x]$	$p[t'/x]q[t'/x]$
$\lambda y.p$	$\lambda y.(p[t/x])$	$\lambda y.(p[t'/x])$

In each case  $s[t/x] \rightarrow_{\parallel} s'[t'/x]$  (in the last two cases use the induction hypothesis).

- ( $\parallel$ -app) Then  $s \equiv pq$  and  $s' \equiv p'q'$ , with  $p \rightarrow_{\parallel} p'$  and  $q \rightarrow_{\parallel} q'$ . But by the induction hypothesis (since  $p$  and  $q$  are subterms of  $s$ ),  $p[t/x] \rightarrow_{\parallel} p'[t'/x]$  and  $q[t/x] \rightarrow_{\parallel} q'[t'/x]$ . Hence by the rule ( $\parallel$ -app),  $s[t/x] \rightarrow_{\parallel} s'[t'/x]$ .
- (abs) Exercise.
- ( $\parallel$ - $\beta$ ) Then  $s \equiv (\lambda y.p)q$  and  $s' \equiv p'[q'/y]$  with  $p \rightarrow_{\parallel} p'$  and  $q \rightarrow_{\parallel} q'$ . By the induction hypothesis  $p[t/x] \rightarrow_{\parallel} p'[t'/x]$  and  $q[t/x] \rightarrow_{\parallel} q'[t'/x]$ . So

$$\begin{aligned}
s[t/x] &\equiv ((\lambda y.p)q)[t/x] \\
&\equiv (\lambda y.p[t/x])(q[t/x]) \\
&\rightarrow_{\parallel} p'[t'/x][q'[t'/x]/y] && \text{by } (\parallel\text{-}\beta) \\
&\equiv (p'[q'/y])[t'/x] && \text{by Exercise 1.5} \\
&\equiv s'[t'/x]
\end{aligned}$$

■

**Lemma 2.5.2**  $\rightarrow_{\parallel}$  satisfies the diamond property. That is, for all  $s$ , if  $s \rightarrow_{\parallel} s_1$  and  $s \rightarrow_{\parallel} s_2$  then there is some  $t$  such that  $s_1 \rightarrow_{\parallel} t$  and  $s_2 \rightarrow_{\parallel} t$ .

**Proof** We prove the result by induction on  $s$ , and by case analysis on the rules showing  $s \rightarrow_{\parallel} s_1$  and  $s \rightarrow_{\parallel} s_2$ :

- ( $s \rightarrow_{\parallel} s_1$  by (refl),  $s \rightarrow_{\parallel} s_2$  by any rule)  
Then  $s \equiv s_1$ , so we can take  $t \equiv s_2$ .
- ( $s \rightarrow_{\parallel} s_1$  by ( $\parallel$ -app),  $s \rightarrow_{\parallel} s_2$  by ( $\parallel$ -app))  
So  $s \equiv pq$ ,  $s_1 \equiv p_1q_1$ ,  $s_2 \equiv p_2q_2$ , with  $p \rightarrow_{\parallel} p_1$ ,  $p \rightarrow_{\parallel} p_2$ ,  $q \rightarrow_{\parallel} q_1$  and  $q \rightarrow_{\parallel} q_2$ .

But by applying the induction hypothesis to  $p$ , and again to  $q$ , there are terms  $p_3$  and  $q_3$  with  $p_1 \rightarrow_{\parallel} p_3$ ,  $p_2 \rightarrow_{\parallel} p_3$ ,  $q_1 \rightarrow_{\parallel} q_3$  and  $q_2 \rightarrow_{\parallel} q_3$ . Hence take  $t \equiv p_3 q_3$  and the result follows by ( $\parallel$ -app).

- ( $s \rightarrow_{\parallel} s_1$  by ( $\parallel$ -app),  $s \rightarrow_{\parallel} s_2$  by ( $\parallel$ - $\beta$ ))

So  $s \equiv (\lambda x.p)q$ ,  $s_1 \equiv (\lambda x.p_1)q_1$ ,  $s_2 \equiv p_2[q_2/x]$  with  $p \rightarrow_{\parallel} p_1$ ,  $p \rightarrow_{\parallel} p_2$ ,  $q \rightarrow_{\parallel} q_1$  and  $q \rightarrow_{\parallel} q_2$ .

By applying the induction hypothesis to  $p$ , and again to  $q$ , there are terms  $p_3$  and  $q_3$  with  $p_1 \rightarrow_{\parallel} p_3$ ,  $p_2 \rightarrow_{\parallel} p_3$ ,  $q_1 \rightarrow_{\parallel} q_3$  and  $q_2 \rightarrow_{\parallel} q_3$ . Take  $t \equiv p_3[q_3/x]$ . By  $\parallel$ - $\beta$ ,  $s_1 \equiv (\lambda x.p_1)q_1 \rightarrow_{\parallel} t$ , and by the Substitution Lemma 2.5.1  $s_2 \equiv p_2[q_2/x] \rightarrow_{\parallel} t$ .

- ( $s \rightarrow_{\parallel} s_1$  by ( $\parallel$ - $\beta$ ),  $s \rightarrow_{\parallel} s_2$  by ( $\parallel$ - $\beta$ ))

Exercise.

- ( $s \rightarrow_{\parallel} s_1$  by (abs) — then because  $s$  must be an abstraction, we must have  $s \rightarrow_{\parallel} s_2$  by (abs) also)

Exercise.

- ( $s \rightarrow_{\parallel} s_2$  by (refl),  $s \rightarrow_{\parallel} s_1$  by any rule) or ( $s \rightarrow_{\parallel} s_1$  by ( $\parallel$ - $\beta$ ),  $s \rightarrow_{\parallel} s_2$  by ( $\parallel$ -app))

Symmetrical to one of the above.

■

**Lemma 2.5.3**  $\rightarrow_{\beta}$  is the reflexive transitive closure of  $\rightarrow_{\parallel}$ .

**Proof** Note that

$$\rightarrow_{\beta} \subseteq \rightarrow_{\parallel} \subseteq \rightarrow_{\beta},$$

by Lemma 1.1.1. Taking the reflexive transitive closure of a relation is a monotone (w.r.t. inclusion) operation on relations, and the reflexive transitive closure of  $\rightarrow_{\beta}$  is  $\rightarrow_{\beta}$ , which is already reflexive and transitive. So taking the reflexive transitive closure all three sides of the above inclusions gives the required result. ■

Hence,

**Theorem 2.5.4**  $\beta$ -reduction is Church-Rosser.

## 2.6 Consistency

We can combine the results of this chapter to show that the theory  $\lambda\beta$  is consistent.

**Theorem 2.6.1 (Consistency of  $\lambda\beta$ )** There are terms  $s$  and  $t$  for which  $\lambda\beta \not\vdash s = t$ .

**Proof** Take the standard combinators  $\mathbf{k} \equiv \lambda xy.x$  and  $\mathbf{s} \equiv \lambda xyz.xz(yz)$ . Then

$$\begin{aligned} \lambda\beta \vdash \mathbf{s} = \mathbf{k} &\iff \mathbf{s} =_{\beta} \mathbf{k} && \text{by Theorem 2.2.1} \\ &\iff \exists u. \mathbf{s} \rightarrow_{\beta} u \text{ and } \mathbf{k} \rightarrow_{\beta} u && \text{by Lemma 2.4.1(ii)} \\ &&& \text{because } \beta\text{-reduction is C-R} \end{aligned}$$

But  $\mathbf{s}$  and  $\mathbf{k}$  are distinct normal forms, so this last cannot hold.  $\blacksquare$

The necessary additional results to show that  $\lambda\beta\eta$  is also consistent can be found in the exercises.

If we consider  $\beta$ -reduction to be the computation we perform on terms, then terms *without* normal forms are non-terminating computations. We might therefore wish to say that all terms without a normal form are “undefined” and form a  $\lambda$ -theory which makes them all equal.

**Definition** The  $\lambda$ -theory  $\mathcal{T}_{\text{NF}}$  is defined by

$$\mathcal{T}_{\text{NF}} = \lambda\beta + \{s = t \mid s, t \text{ are closed terms without normal forms}\}.$$

(Caution:  $\mathcal{T}_{\text{NF}}$  sometimes means something else in the literature.) However,

**Lemma 2.6.2**  $\mathcal{T}_{\text{NF}}$  is inconsistent.

**Proof** The terms  $\lambda xy.x\Omega$  and  $\lambda xy.y\Omega$  do not have normal forms. So for any terms  $s$  and  $t$ ,

$$\begin{aligned} \mathcal{T}_{\text{NF}} \vdash s &= \mathbf{k}s\Omega \\ &= (\lambda xy.x\Omega)(\mathbf{k}s)(\mathbf{k}t) \\ &= (\lambda xy.y\Omega)(\mathbf{k}s)(\mathbf{k}t) \\ &= \mathbf{k}t\Omega \\ &= t \end{aligned}$$

hence  $\mathcal{T}_{\text{NF}}$  equates all terms.  $\blacksquare$

(One of the nice properties of the  $\Lambda I$ -calculus – where the set of terms is restricted to  $\Lambda I$ -terms – is that it *is* consistent to equate terms without normal forms. The proof is nontrivial, see [Bar84, 16.1.13].)

In Chapter 3 we will see a different set of terms which can plausibly represent “undefined” computation and which *can* be consistently identified in the full  $\lambda$ -calculus.

## 2.7 Böhm's Theorem

We have seen that  $\mathbf{s}$  and  $\mathbf{k}$  are not equated by  $\lambda\beta$ . In fact, something stronger holds:

**Lemma 2.7.1**  $\lambda\beta + \{\mathbf{s} = \mathbf{k}\}$  is inconsistent.

(This means that no consistent  $\lambda$ -theory can possibly equate  $\mathbf{s}$  and  $\mathbf{k}$ .)

**Proof** We seek to prove that  $\mathbf{i} = t$  for all terms  $t$ , using  $\lambda\beta$  plus the additional axiom  $\mathbf{s} = \mathbf{k}$ :

$$\lambda\beta + \{\mathbf{s} = \mathbf{k}\} \vdash \mathbf{skkit} = \mathbf{kkkit}$$

(by repeated use of the application rule).

The left is equal (just using  $\lambda\beta$ ) to  $\mathbf{ki}(\mathbf{ki})t$ , which is equal to  $t$ . The right is equal to  $\mathbf{kit}$  which is equal to  $\mathbf{i}$ . So  $\mathbf{i} = t$  for any term  $t$ , so all terms are equated with each other by transitivity. ■

Now the choice of terms  $\mathbf{s}$  and  $\mathbf{k}$  in Theorem 2.6.1 was fairly arbitrary. The proof works for any pair of terms in normal form. Also (see Exercise 2.16) the equivalent result holds for  $\lambda\beta\eta$ . So,

**Lemma 2.7.2**

- (i) For any two terms  $s$  and  $t$  in  $\beta$ -normal form and not identical,  $\lambda\beta \not\vdash s = t$ .
- (ii) For any two terms  $s$  and  $t$  in  $\beta\eta$ -normal form and not identical,  $\lambda\beta\eta \not\vdash s = t$ .

What about generalising Lemma 2.7.1? Is there a consistent way to equate some distinct normal forms? Yes – in  $\lambda\beta\eta$  we equate the distinct  $\beta$ -normal forms  $\lambda x.x$  and  $\lambda xy.xy$ . But for terms which are distinct  $\beta\eta$ -normal forms the answer is no. To show this requires the power of **Böhm's Theorem**, a very strong result which comes from syntactical analysis in the  $\lambda$ -calculus.

Recall the two standard combinators

$$\mathbf{t} \equiv \lambda xy.x(\equiv \mathbf{k}) \quad \mathbf{f} \equiv \lambda xy.y$$

**Theorem 2.7.3 (Böhm's Theorem)** Let  $s$  and  $t$  be distinct closed  $\beta\eta$ -normal forms. Then there exist terms  $u_1, \dots, u_n$  such that

$$\begin{aligned} \lambda\beta \vdash su_1u_2 \dots u_n &= \mathbf{t} \\ \lambda\beta \vdash tu_1u_2 \dots u_n &= \mathbf{f} \end{aligned}$$

**Proof** Actually not difficult to comprehend, but too long for this course.

A proof can be found in [Bar84, 10.3]; the original was in [Böh68]. ■

Böhm's Theorem is a **separation** result, giving sufficient (but not necessary) conditions for two terms to be completely distinguished in the equational theory of the  $\lambda$ -calculus. A related result is the Genericity Lemma, which shows that some terms are impossible to separate, and we will come to this in Chapter 3.

Separable terms cannot consistently be equated:

**Theorem 2.7.4** If  $s$  and  $t$  are distinct closed  $\beta\eta$ -normal forms then  $\lambda\beta + \{s = t\}$  is inconsistent (hence the same holds for any  $\lambda$ -theory which equates  $s$  and  $t$ ).

**Proof** If  $s$  and  $t$  are distinct normal forms then there are terms  $u_1, \dots, u_n$  such that  $\lambda\beta \vdash s\vec{u} = \mathbf{t}^2$  and  $\lambda\beta \vdash t\vec{u} = \mathbf{f}$ , by Böhm's Theorem.

Thus for any terms  $p$  and  $q$ ,  $\lambda\beta + \{s = t\} \vdash p = \mathbf{t}pq = s\vec{u}pq = t\vec{u}pq = \mathbf{f}pq = q$ . ■

(It is not hard to extend this result to terms with free variables, and there is a slightly weaker result for terms which *have* distinct  $\beta\eta$ -normal forms but are not necessarily themselves in normal form.)

So we have shown that  $\lambda\beta$  and  $\lambda\beta\eta$  do not equate any terms which are different normal forms, and to do so would be inconsistent. Likewise we have shown that to equate all terms without normal form is inconsistent. What *can* we consistently equate? Since  $\lambda\beta\eta = \lambda\beta + \{\lambda xy.xy = \lambda x.x\}$ , and the following exercises will show that  $\lambda\beta\eta$  is consistent, we have shown that you can consistently equate  $\lambda xy.xy$  and  $\lambda x.x$  in  $\lambda\beta$ , and more generally it will be consistent to equate any terms equated by  $\lambda\beta\eta$ .

Actually there are many interesting answers. For example, the theory  $\lambda\beta + \{\lambda x.x = \mathbf{\Omega}\}$  is consistent — perhaps a surprising fact, and quite tough to prove. We will see another interesting answer in Chapter 3.

## Computational Practice

**2.a** Which of the following are true?

- (i)  $\lambda x.x \beta \lambda x.x$ ;
- $\lambda x.x \rightarrow_{\beta} \lambda x.x$ ;
- $\lambda x.x \twoheadrightarrow_{\beta} \lambda x.x$ .

<sup>2</sup> $s\vec{u}$  is common shorthand for  $su_1u_2 \dots u_n$ . Note the implicit bracketing here:  $s\vec{u}$  is not  $s(\vec{u})$ !

- (ii)  $(\lambda xy.x)pq \beta p$ ;  
 $(\lambda xy.x)pq \rightarrow_{\beta} p$ ;  
 $(\lambda xy.x)pq \twoheadrightarrow_{\beta} p$ .
- (iii)  $(\lambda x.x)((\lambda y.p)q) \beta (\lambda x.x)q$ ;  
 $(\lambda x.x)((\lambda y.p)q) \rightarrow_{\beta} (\lambda x.x)q$ ;  
 $(\lambda x.x)((\lambda y.p)q) \twoheadrightarrow_{\beta} (\lambda x.x)q$ .
- (iv)  $\Omega \beta \Omega$ ;  
 $\Omega \rightarrow_{\beta} \Omega$ ;  
 $\Omega \twoheadrightarrow_{\beta} \Omega$ .

**2.b** Determine *all* redexes in each of:

- (i)  $(\lambda x.x)(\mathbf{k}\mathbf{k})$ .
- (ii)  $(\lambda xy.(\lambda z.z)x)\Omega$ .
- (iii)  $\lambda x.(\lambda y.(\lambda z.(\lambda p.p)z)y)x$ .
- (iv) **yii**.

**2.c** Compute the  $\beta$ -normal form of:

- (i)  $\lambda x.(\lambda y.(\lambda z.(\lambda p.p)z)y)x$ .
- (ii)  $\mathbf{k}x\Omega$ .
- (iii)  $\mathbf{s}\mathbf{k}\mathbf{k}$ .
- (iv)  $\mathbf{s}\mathbf{k}\Omega$ .

**2.d** Draw all possible  $\beta$ -reduction paths from:

- (i)  $(\lambda xy.yy)\Omega\mathbf{i}$ .
- (ii)  $((\lambda y.yy)\mathbf{k})((\lambda y.yy)\mathbf{k})$ .
- (iii)  $\lambda x.(\lambda y.(\lambda z.(\lambda p.p)z)y)x$ .

**2.e** Which of the following have  $\beta$ -normal forms?

- (i)  $(\lambda x.xx)((\lambda xy.x)(\lambda y.y)(\lambda z.zz))$ .
- (ii)  $\mathbf{k}\mathbf{k}\mathbf{k}\mathbf{k}\mathbf{k}\mathbf{k}\mathbf{k}\mathbf{k}\mathbf{k}$ .
- (iii)  $(\lambda x.xx)(\lambda y.y)(\lambda x.xx)$ .
- (iv)  $(\lambda xy.xyy)(\lambda y.y)(\lambda x.xx)$ .

## Exercises

**2.1** Prove Lemma 2.1.2.

**2.2** Give pairs of terms to show that the following inclusions are strict:

$$\rightarrow_{\beta} \subset \rightarrow_{\beta}^+ \subset \twoheadrightarrow_{\beta} \subset =_{\beta}$$

**2.3** Show that the following terms have a  $\beta$ -normal form. Are they strongly normalizable?

- (i)  $(\lambda yz.zy)((\lambda x.xx)(\lambda x.xx))(\lambda wx.x)$
- (ii)  $(\lambda ab.ba)(\mathbf{k}\mathbf{k})(\lambda y.yy)$

**2.4** Show that  $\Omega$  is the only term  $t$  such that  $\Omega \twoheadrightarrow_{\beta} t$ . Show also that for all terms  $f$ ,  $\theta f \twoheadrightarrow_{\beta} f(\theta f)$ .

**2.5** Give two  $\lambda$ -terms which are:

- (i) in  $\beta$ -normal form.
- (ii) not in  $\beta$ -normal form but strongly normalizable.
- (iii) normalizable but not strongly normalizable.
- (iv) not normalizable.

**2.6** Prove Lemma 2.3.2.

**2.7** Find a notion of reduction  $R$  (on some arbitrary set, not necessarily  $\Lambda$ ) such that  $\rightarrow_R$  has the unique normal form property, but is not Church-Rosser.

Show that, if  $\rightarrow_R$  is strongly normalizing and has the unique normal form property then it is Church-Rosser. Is the statement still valid if  $\rightarrow_R$  is only weakly normalizing?

**2.8** Show that  $\beta$ -reduction implements  $\lambda\beta$ . That is,  $\lambda\beta \vdash s = t$  if and only if  $s =_{\beta} t$ .

[Hint: use Lemma 1.1.1]

**2.9** Give rules which define recursively the set of  $\beta$ -normal forms. Do the same for  $\beta\eta$ -normal forms.

**2.10** Fill in the missing cases in Lemmas 2.5.1 and 2.5.2.

**2.11** *Without* using Böhm's Theorem, show that:

- (i)  $\lambda\beta + \{\mathbf{i} = \mathbf{k}\}$  is inconsistent.
- (ii)  $\lambda\beta + \{\mathbf{i} = \mathbf{s}\}$  is inconsistent.
- (iii)  $\lambda\beta + \{xy = yx\}$  is inconsistent.



[Hint: By applying both sides to appropriate terms, show that  $\mathbf{i} = s$  for all terms  $s$ .]

The next 5 exercises extend the proof of the Church-Rosser property to  $\beta\eta$ -reduction, and show consistency of  $\lambda\beta\eta$ .

Let  $\rightarrow_1$  and  $\rightarrow_2$  be binary relations over  $\Lambda$ . We say that  $\rightarrow_1$  and  $\rightarrow_2$  **commute** if  $s \rightarrow_1 t_1$  and  $s \rightarrow_2 t_2$  implies that there exists  $t$  with  $t_1 \rightarrow_2 t$  and  $t_2 \rightarrow_1 t$ .

[Note that a relation  $\rightarrow$  satisfies the diamond property if and only if it commutes with itself.]

**2.12** [Hindley and Rosen] Let  $\rightarrow_1$  and  $\rightarrow_2$  be binary relations over  $\Lambda$  which commute, and suppose that both satisfy the diamond property. Define  $\rightarrow_{12}$  to be the union of the two relations, and  $\twoheadrightarrow_{12}$  the transitive closure of  $\rightarrow_{12}$ . Show that  $\twoheadrightarrow_{12}$  satisfies the diamond property.

**2.13** Let  $\rightarrow_\eta$  be the one-step  $\eta^{\text{red}}$ -reduction. Show that  $\twoheadrightarrow_\beta$  and  $\twoheadrightarrow_\eta$  commute.

[Hint: It suffices to show that if  $s \twoheadrightarrow_\beta s_1$  and  $s \twoheadrightarrow_\eta s_2$  then there is  $t$  such that  $s_1 \twoheadrightarrow_\eta t$  and  $s_2 \twoheadrightarrow_\beta t$ , then do a diagram chase.]

**2.14** Show that  $\eta^{\text{red}}$  is Church-Rosser.

**2.15** Use the results of Exercises 2.12 to 2.14 to show that the notion of reduction  $\beta\eta$  is Church-Rosser.

**2.16** Show that the notion of reduction  $\beta\eta$  implements  $\lambda\beta\eta$ . Deduce that  $\lambda\beta\eta$  is consistent.

**\*2.17** Add a constant  $\delta_C$  to the terms of the  $\lambda$ -calculus, using the same formation rules as for  $\Lambda$ . Call these terms  $\Lambda\delta_C$ . Extend the notion of  $\beta$ -reduction to  $\Lambda\delta_C$  in the obvious way.

Define the set  $S$  of  $\beta\delta_C$ -**normal forms** to be the set of *closed*  $\beta$ -normal forms of  $\Lambda\delta_C$  which do not contain a subterm  $\delta_C pq$ .

Define the notion of reduction  $\delta$  in the following way:

$$\begin{aligned} \delta &= \{ \langle \delta_C ss, \mathbf{t} \rangle, \text{ for } s \text{ in } S \} \\ &\cup \{ \langle \delta_C st, \mathbf{f} \rangle, \text{ for distinct terms } s \text{ and } t \text{ in } S \}. \end{aligned}$$

The notion of reduction  $\beta\delta$  is the union of  $\beta$  and  $\delta$ . Show that  $\beta\delta$  is Church-Rosser.



## Chapter 3

# Reduction Strategies

---

**Reading:** [Bar84] 11.4, 13.2, 8.3, 16.1, 16.2; [BB94] 4;  
[Han94] 3.6, 3.7

---

Recall that it is possible for a term of the untyped  $\lambda$ -calculus to  $\beta$ -reduce in one step to distinct terms, and hence to a potential plethora of reduction paths. Thanks to the Church-Rosser property, we know that these paths can only reach one normal form, but that does not guarantee that all paths reach the normal form, and of course some paths might be much shorter than others. If we are modelling computation as reduction, the latter corresponds to less or more efficient computation.

Were the  $\lambda$ -calculus to be implemented on a computer, one presumes that reduction would proceed deterministically, and we would want to find normal forms wherever possible. The study of **reduction strategies** is quite complex, and it is difficult to avoid very technical discussions of the nature of  $\beta$ -redexes, and what they leave behind when they reduce (chapters 11–13 of [Bar84] provide an in-depth study of reduction strategies). In this chapter we will be selective and avoid the worst of the details, relying on one key lemma which we will not prove. We aim to prove that a certain  $\beta$ -reduction strategy will always find normal forms, when they exist. Along the way we learn something new about “undefined”, in the realm of computation as reduction.

### 3.1 Reduction Strategies

A general definition from term rewriting is as follows:

**Definition** A **reduction strategy** is a binary relation  $\rightarrow$  satisfying: for all  $s$ , if  $s \rightarrow t_1$  and  $s \rightarrow t_2$  then  $t_1 \equiv t_2$ .

A reduction strategy can also be thought of as a (partial) function, mapping each term to a unique reduct, if it has a reduct at all. Reduction strategies automatically have the unique normal form property.

Generally, a reduction strategy is *not* compatible with the formation rules for terms — reduction will not proceed inside an arbitrary context (else reducts would not be unique).

Here we will only be interested in reduction strategies for  $\beta$ -reduction, i.e. reduction strategies  $\rightarrow_C \rightarrow_\beta$ . One can imagine quite complex strategies, for example:

- (i) Reduce the first redex you see, from the left of the term (or the right);
- (ii) Make some complexity measure on redexes, and reduce the most “complex” each time (this idea is used in Turing’s proof of weak normalisation for the simply typed  $\lambda$ -calculus [Tur42]);
- (iii) Reduce whichever redex leads to the shortest path to a normal form, if there is one (otherwise pick a redex at random)<sup>1</sup>.

Here are two important, related, reduction strategies:

### Definition

- (i) The binary relation  $\xrightarrow{h}$ , called **head reduction**, on terms is defined by

$$\lambda x_1 \dots x_m. (\lambda y. t) u s_1 \dots s_n \xrightarrow{h} \lambda x_1 \dots x_m. t[u/y]s_1 \dots s_n$$

(where  $m, n \geq 0$ ).

$\xrightarrow{h}$  is the reflexive transitive closure of  $\xrightarrow{h}$ .

- (ii) The binary relation  $\xrightarrow{l}$ , called **leftmost reduction**, on terms is defined by  $s \xrightarrow{l} t$  if  $s \rightarrow_\beta t$  and the redex  $(\lambda x. p)q$  which is reduced in  $s$  is the leftmost amongst all redexes in  $s$  (measuring the position of a redex by its beginning).

$\xrightarrow{l}$  is the reflexive transitive closure of  $\xrightarrow{l}$ .

Both  $\xrightarrow{h}$  and  $\xrightarrow{l}$  are clearly reduction strategies for  $\beta$ -reduction. Note that every head reduction is a leftmost reduction but not necessarily vice versa. Also, the normal forms of  $\xrightarrow{l}$  are precisely the same as the normal forms of  $\rightarrow_\beta$ , since if there is a  $\beta$ -reduction from a term then there is a leftmost  $\beta$ -reduction. But the normal forms of  $\xrightarrow{h}$  are a proper superset of the normal forms of  $\rightarrow_\beta$ , for example  $\lambda x. x((\lambda y. y)x)$  does not match the left hand side of the definition of  $\xrightarrow{h}$  and so is a  $\xrightarrow{h}$ -normal form. Rather than  $\xrightarrow{h}$ -normal form we tend to say **head normal form** or **hnf**.

---

<sup>1</sup>This reduction strategy is all very well, but it turns out to be computationally as complex to work out which is the best redex to reduce as it is to compute the normal form anyway. The study of **optimal reduction** is interesting and difficult.

In the term  $\lambda x_1 \dots x_m. (\lambda y. t) u s_1 \dots s_n$  the subterm  $(\lambda y. t) u$  is called the **head redex**. The previous paragraph is equivalent to: every leftmost redex is a head redex, but not every term which has a redex has a head redex.

Another useful definition is:

**Definition** Given a binary relation  $\rightarrow$  and a term  $s$  a **reduction sequence** for  $s$  is a (finite or infinite) sequence  $s \rightarrow s_1 \rightarrow s_2 \rightarrow \dots$ .

If  $\rightarrow$  is a reduction strategy then each term has a unique maximal reduction sequence (terminating in the unique  $\rightarrow$ -normal form of  $s$ , if it has one, and infinite otherwise).

Now let us examine  $\xrightarrow{h}$  a little more closely. Note that any term  $s$  must be of the form

- (i)  $s \equiv \lambda x_1 \dots x_m. y s_1 \dots s_n$  for some variable  $y$  and  $n, m \geq 0$ , or
- (ii)  $s \equiv \lambda x_1 \dots x_m. (\lambda y. t) u s_1 \dots s_n$  for some variable  $y$  and  $n, m \geq 0$ ,

simply by peeling off as many abstractions from the outside of  $s$  as possible, then as many applications as possible (looking at the left of each application in turn), and seeing whether what is left is a variable or an abstraction. (This is better visualised by drawing construction trees.)

Terms of type (i) are the normal forms of  $\xrightarrow{h}$ ; terms of type (ii) are those which  $\xrightarrow{h}$ -reduce. The principle is that  $\xrightarrow{h}$  is a particular type of  $\beta$ -reduction which *can* proceed on the left of an application inside an abstraction, but *cannot* proceed on the right of an application, nor inside an abstraction which is itself on the left of an application.

It is worthwhile to have a symbol for  $\beta$ -reductions other than  $\xrightarrow{h}$ :

**Definition** We write  $s \xrightarrow{i} t$  if  $s \rightarrow_{\beta} t$  but  $s \not\xrightarrow{h} t$ . ( $\xrightarrow{i}$  is known as **internal reduction**.)  $\xrightarrow{i}$  is the reflexive transitive closure of  $\xrightarrow{i}$ .

(Of course, this is not a reduction strategy.) Notice that if  $s \xrightarrow{i} t$  and  $s$  has a head redex then  $t$  still has a head redex: internal reduction cannot destroy head redexes (but it might alter them, reducing one of their components). Now we must make use of a result about head reduction, which will not be proved here for reasons of space. It involves another theorem which we do not cover in this course (the Finiteness of Developments Theorem) and a full proof can be found for example in [Bar84, §11.4].

**Lemma 3.1.1 (Advancement of Head Reduction)** If  $s \rightarrow_{\beta} t$  then there exists a term  $u$  such that  $s \xrightarrow{h} u \xrightarrow{i} t$ . i.e. we can find a reduction path to the same result with all head reductions performed first, and all internal reductions performed later.

**Proof** Omitted and non-examinable. ■

### 3.2 Standard Reductions and Standardization

Using this property of  $\xrightarrow{h}$  and  $\xrightarrow{i}$  we can prove a powerful result about  $\xrightarrow{l}$ , but first yet more definitions (this one rather informal, although it can be made precise):

**Definition** A  $\beta$ -reduction sequence is **standard**, written  $s \twoheadrightarrow_s t$ , if the following is true:

“Write the term  $s$  in black. Suppose  $s_i \rightarrow s_{i+1}$  by  $\beta$ -reducing a redex  $R = (\lambda x.p)q$  in  $s_i$ . Then in  $s_{i+1}$  all the  $\lambda$ 's to the left of this  $\lambda$  of  $R$ 's reduct turn red, as well as any red  $\lambda$ 's inherited from  $s_i$ .

In the reduction sequence, only redexes  $(\lambda x.p)q$  with a black  $\lambda$  are allowed to reduce; red  $\lambda$ 's are ‘frozen’.”

A standard reduction reduces redexes from left to right. For example, here are two  $\beta$ -reduction paths from one term to its  $\beta$ -normal form:

- $\lambda x.(\lambda y.(\lambda z.z)yy)t \rightarrow_{\beta} \lambda x.(\lambda y.yy)t \rightarrow_{\beta} \lambda x.tt$
- $\lambda x.(\lambda y.(\lambda z.z)yy)t \rightarrow_{\beta} \lambda x.(\lambda z.z)tt \rightarrow_{\beta} \lambda x.tt$

The first is not standard, the second is. It is important to note that “standard” is a property of a whole reduction sequence, not of the individual steps of the reduction sequence. (Individually, every one-step  $\beta$ -reduction is standard, because initially all terms are coloured “black”). So “standard” reduction is not a reduction strategy or even a one-step reduction in its own right.

#### Lemma 3.2.1

- (i) Any subsequence of a standard reduction is standard.
- (ii) Every  $\xrightarrow{l}$ -reduction sequence is standard.
- (iii) Every  $\xrightarrow{h}$ -reduction sequence is standard.
- (iv) If  $s \twoheadrightarrow_s t$ , with  $t$  a  $\beta$ -normal form, then all the reductions in the given standard reduction sequence are leftmost reductions.

**Proof** (i) and (ii) are obvious. For (iii), the only  $\lambda$ 's which get turned red when

$$\lambda x_1 \dots x_m.(\lambda y.t)us_1 \dots s_n \xrightarrow{h} \lambda x_1 \dots x_m.t[u/y]s_1 \dots s_n$$

are those for  $x_1, \dots, x_m$ , and these abstractions can never become part of a redex because there is no application of which they are the left-hand side.

For (iv), argue by contradiction. If a leftmost redex  $(\lambda x.p)q$  is not reduced at some step of the standard reduction sequence, its  $\lambda$  is turned red and

frozen. The terms  $p$  and  $q$  might reduce, to  $p'$  and  $q'$ , later in the standard reduction sequence. But the the frozen redex  $(\lambda x.p')q'$  must remain all the way to the end, contradicting the assumption that  $t$  was a normal form. ■

Now (admittedly using Lemma 3.1.1, which we have not proved here) we can show a classical result of the untyped  $\lambda$ -calculus, which has far-reaching implications:

**Theorem 3.2.2 (Standardization)** If  $s \rightarrow_{\beta} t$  then  $s \rightarrow_s t$ . i.e. standard reduction sequences suffice.

Note: this is to say that there exists a standard reduction from  $s$  to  $t$ , not that the  $\beta$ -reduction sequence given is necessarily standard.

**Proof** By induction on the length of the term  $t$ . By Lemma 3.1.1 there exists a term  $u$  such that  $s \xrightarrow{h} u \xrightarrow{i} t$ .

For the base case of the induction,  $t$  is of length 1 and so must be a variable hence (as internal reductions always produce something more than just a single variable)  $u \equiv t$  and we are done, by Lemma 3.2.1(iii).

For the inductive case, suppose the result for terms shorter than  $t$ . Write  $t$  in the form  $t \equiv \lambda x_1 \dots x_m. t_0 t_1 \dots t_n$ . Then  $u$  must have been of the form  $u \equiv \lambda x_1 \dots x_m. u_0 u_1 \dots u_n$  with  $u_i \rightarrow_{\beta} t_i$ .

But by the induction hypothesis there are standard reductions  $u_i \rightarrow_s t_i$ , since each  $t_i$  is shorter than  $t$ , and it follows that

$$\begin{aligned} s & \xrightarrow{h} \lambda x_1 \dots x_n. u_0 u_1 \dots u_n \\ & \rightarrow_s \lambda x_1 \dots x_n. t_0 u_1 \dots u_n \\ & \rightarrow_s \lambda x_1 \dots x_n. t_0 t_1 \dots u_n \\ & \quad \vdots \\ & \rightarrow_s \lambda x_1 \dots x_n. t_0 t_1 \dots t_n \equiv t \end{aligned}$$

is a standard reduction from  $s$  to  $t$ . ■

It is fairly easy to see that there can only be at most one standard reduction from  $s$  to  $t$ . So it makes sense to talk about “the” standard reduction from  $s$  to  $t$ , if there is any  $\beta$ -reduction path from  $s$  to  $t$ .

A powerful result which can be derived from the Standardization Theorem is the following. It has practical implications for the design of functional programming languages.

**Theorem 3.2.3** If  $s \rightarrow_{\beta} t$ , where  $t$  is a  $\beta$ -normal form, then  $s \xrightarrow{l} t$ . i.e. leftmost reduction is sufficient to find the normal form of any term that has one.

(We say that leftmost reduction is **normalising**.)

**Proof**

If  $s \twoheadrightarrow_{\beta} t$  then there is a standard reduction sequence  $s \twoheadrightarrow_s t$ , by the Standardization Theorem. Since  $t$  is a  $\beta$ -normal form we can use Lemma 3.2.1(iv) to deduce that this standard reduction sequence is a leftmost reduction sequence. ■

### 3.3 More on Head Normal Forms

We introduced head reduction because it was necessary for the proof of normalisation of leftmost reduction. However head reduction, and the set of terms which has no head normal form, have fascinating mathematical aspects in their own right. Furthermore, programming languages based on the  $\lambda$ -calculus tend to use reductions more similar to head reduction than full leftmost reduction (for reasons beyond the scope of this course).

Recall that every  $\beta$ -normal form is also a head normal form, but not vice versa. There are terms which do have a head normal form, but not a  $\beta$ -normal form (for example  $\lambda x.x\Omega$ ). Consider the following definition, couched solely in terms of the equational theory  $\lambda\beta$  and which has no apparent connection with head reduction:

#### Definition

A closed term  $s$  is **solvable** if there exist closed terms  $t_1, \dots, t_n$  such that  $\lambda\beta \vdash st_1 \dots t_n = \mathbf{i}$ .

A term which is not solvable is called **unsolvable**.

For example,  $\lambda xy.xy$  is solvable:  $(\lambda xy.xy)\mathbf{i} = \mathbf{i}$ .  $\mathbf{s}$  is solvable:  $\mathbf{s}\mathbf{i}\mathbf{i}\mathbf{i} = \mathbf{i}$ .  $\mathbf{y}$  is solvable:  $\mathbf{y}(\mathbf{k}\mathbf{i}) = (\mathbf{k}\mathbf{i})(\mathbf{y}(\mathbf{k}\mathbf{i})) = \mathbf{i}$ .  $\lambda x.x\Omega$  is solvable:  $(\lambda x.x\Omega)(\mathbf{k}\mathbf{i}) = \mathbf{i}$ .

(This definition is extended to terms with free variables by abstracting away all free variables, prior to application to  $t_1, \dots, t_n$ , but we will not worry about this detail in this course.)

#### Lemma 3.3.1

- (i) If  $st$  is solvable then  $s$  is solvable.
- (ii)  $\Omega$  is unsolvable.

**Proof** (i) If  $st$  is solvable then there are terms  $u_1, \dots, u_n$  such that  $\lambda\beta \vdash stu_1, \dots, u_n = \mathbf{i}$ . Setting  $v_1 \equiv t$  and  $v_{i+1} \equiv u_i$  we see that  $\lambda\beta \vdash sv_1, \dots, v_{n+1} = \mathbf{i}$ .



(ii) If  $\Omega\vec{t} = \mathbf{i}$  then by the Church-Rosser property of  $\beta$ -reduction,  $\Omega\vec{t} \rightarrow_{\beta} \mathbf{i}$ . But the only term  $u$  such that  $\Omega \rightarrow_{\beta} u$  is  $\Omega$  itself. This implies that  $\Omega\vec{t} \rightarrow_{\beta} u$  only if  $u \equiv \Omega\vec{t}'$  for some  $\vec{t}'$ . Hence  $\Omega\vec{t} \not\rightarrow_{\beta} \mathbf{i}$  for any  $\vec{t}$ . ■

Why is the term  $\mathbf{i}$  is chosen for the definition? Because if  $s$  is solvable then for any  $u$  we can “solve” the equation  $s\vec{t} = u$ .

Now it is easy to prove:

**Lemma 3.3.2** If  $s$  is a closed term which has a head normal form, then  $s$  is solvable.

**Proof** Let  $t$  be the head normal form of  $s$ , so at least  $\lambda\beta \vdash s = t$ . As a head normal form,  $t$  must be of the form  $\lambda x_1 \dots x_m. y t_1 \dots t_n$  for some variable  $y$  and  $n, m \geq 0$ . Since  $s$  is closed, so is  $t$  ( $\beta$ -reductions cannot create free variables) so  $y$  is  $x_i$ , for some  $i \leq m$ . Let  $u = \lambda z_1 \dots z_n. \mathbf{i}$ . Then  $\lambda\beta \vdash s \underbrace{u \dots u}_m = u t_1 \dots t_n = \mathbf{i}$ . ■

In fact the converse is also true. See Exercises 3.5 to 3.7 for the proof. So we will use the terminology “unsolvable” and “has no hnf” interchangeably.

Note: strictly speaking, some of the proofs in this section need to be extended to deal with free variables.

Recall that it is inconsistent to equate all terms without a  $\beta$ -normal form. This seems incompatible with our slogan that computation is reduction, because terms without a normal form represent undefined, and we would prefer to have undefined be a unique thing. However one can argue that *some* of the terms without a normal form are nevertheless computationally significant (as a trivial example, any term which has  $\mathbf{ki}\Omega$  as a subterm cannot have a  $\beta$ -normal form, but the subterm  $\Omega$  is really irrelevant to the rest of the term).

The set of unsolvable terms (which, we have said, is the same as the set of terms without head normal forms) should really be considered the undefined terms, because they are genuinely computationally insignificant. The following result is a very powerful characterisation of the unsolvable terms:

**Lemma 3.3.3 (Genericity Lemma)** Let  $s, t$  be terms with  $s$  unsolvable and  $t$  having a normal form. Then for all unary contexts  $C$ ,

$$C[s] = t \implies \text{for all terms } u, C[u] = t.$$

**Proof** Difficult. An ingenious proof, which uses a topology on terms (!)

in which normal forms are isolated points and unsolvable terms compactification points, can be found in [Bar84, §14.3]. ■

The moral is that if a computation proceeds involving an unsolvable term, and that computation terminates, the unsolvable term must have been “ignored”. So we try to form a  $\lambda$ -theory which equates all unsolvable terms:

**Definition** The theory  $\mathcal{H}$  is defined by

$$\mathcal{H} = \lambda\beta + \{s = t \mid s, t \text{ are closed unsolvable terms}\}.$$

So  $\mathcal{H}$  is the least (w.r.t. inclusion)  $\lambda$ -theory which equates all unsolvable terms. However it is not at all easy to show that  $\mathcal{H}$  is consistent. We use another, equally interesting,  $\lambda$ -theory to study  $\mathcal{H}$ .

**Definition** The theory  $\mathcal{H}^*$  is defined by

$$\mathcal{H}^* = \{s = t \mid \text{for all unary contexts } C, \\ C[s] \text{ is solvable if and only if } C[t] \text{ is solvable}\}.$$

Although it is quite easy to see that  $\mathcal{H}^*$  does not equate all terms, it not entirely simple to see that it is a  $\lambda$ -theory!

**Lemma 3.3.4**  $\mathcal{H}^*$  is a consistent  $\lambda$ -theory.

**Proof**

We must show that  $\lambda\beta + \mathcal{H}^* = \mathcal{H}^*$ , namely that equality under  $\mathcal{H}^*$  is an equivalence relation, compatible with the formation rules of the  $\lambda$ -calculus, and includes  $\beta$ -conversion.

That equality under  $\mathcal{H}^*$  is an equivalence relation is obvious.

To see that it is compatible with the term formation rules, suppose that  $\mathcal{H}^* \vdash s = t$ . Then for all contexts  $C[X]$ ,  $C[s]$  is solvable if and only if  $C[t]$  is solvable. Take any context  $D[X]$  and let  $C[X] \equiv D[uX]$ . Then

$$\begin{aligned} D[us] \text{ is solvable} &\iff C[s] \text{ is solvable} \\ &\iff C[t] \text{ is solvable} \\ &\iff D[ut] \text{ is solvable} \end{aligned}$$

hence  $\mathcal{H}^* \vdash us = ut$ . Repeating this argument with  $C[X] \equiv D[Xu]$  and  $C[X] \equiv D[\lambda x.X]$  completes the proof.

Finally, suppose that  $\lambda\beta \vdash s = t$  by the  $\beta$ -rule. Let  $C$  be any context (we assume without loss of generality that it has no free variables) then

$$\begin{aligned} C[s] \text{ is solvable} &\iff \exists u_1, \dots, u_m \text{ such that } C[s]\vec{u} = \mathbf{i} \\ &\iff \exists u_1, \dots, u_m \text{ such that } C[t]\vec{u} = \mathbf{i} \\ &\quad (\text{since } \lambda\beta \vdash s = t) \\ &\iff C[t] \text{ is solvable} \end{aligned}$$

hence  $\mathcal{H}^* \vdash s = t$ .

It is simple to see that  $\mathcal{H}^*$  is consistent. Take  $s \equiv \mathbf{i}$ ,  $t \equiv \mathbf{\Omega}$  and  $C[X] \equiv X$ . Then  $C[s] \equiv s$  is solvable and  $C[t] \equiv t$  is unsolvable. Hence  $\mathcal{H}^* \not\vdash s = t$ . ■

Now we can deduce that  $\mathcal{H}$  is consistent, although we need the power of the Genericity Lemma to do so.

**Theorem 3.3.5**

- (i)  $\mathcal{H} \subseteq \mathcal{H}^*$ .
- (ii) Hence  $\mathcal{H}$  is consistent.

**Proof** (i) Since  $\mathcal{H}^*$  is a  $\lambda$ -theory we only need to show that  $\mathcal{H}^* \vdash s = t$  for all (closed) unsolvable terms  $s$  and  $t$ .

Let  $s$  and  $t$  be closed unsolvable terms and  $C$  any unary context, again supposing wlog that  $C$  has no free variables. Then

$$\begin{aligned} C[s] \text{ is solvable} &\iff \text{for some } u_1, \dots, u_m, C[s]\vec{u} = \mathbf{i} \\ &\iff \text{for some } u_1, \dots, u_m, C[t]\vec{u} = \mathbf{i} \\ &\quad \text{by the Genericity Lemma} \\ &\iff C[t] \text{ is solvable} \end{aligned}$$

Hence  $\mathcal{H}^* \vdash s = t$ .

- (ii) Since  $\mathcal{H}^*$  does not equate all terms, neither does  $\mathcal{H}$  by (i). ■

Any  $\lambda$ -theory which equates all unsolvable terms is called **sensible**, and a  $\lambda$ -theory which does not equate an unsolvable term with a solvable term is called **semi-sensible**. (Exercise 3.12 shows that every consistent sensible theory is semi-sensible).

## Computational Practice

**3.a** Compute leftmost reduction sequences from each of the following. Which have  $\beta$ -normal forms?

- (i)  $(\lambda x.xx)\mathbf{k}\mathbf{\Omega}$ .
- (ii)  $(\lambda xy.xxy)(\lambda xy.xxy)((\lambda xy.xyy)(\lambda xy.xyy))$ .
- (iii)  $(\lambda x.xx)(\lambda y.\mathbf{k}(yyy)\mathbf{i})$ .

**3.b** Which of the following terms have a hnf?

- (i)  $\lambda x.\mathbf{\Omega}$ .
- (ii)  $\lambda xy.xxy\mathbf{\Omega}$ .

- (iii)  $\lambda x.(\lambda y.yyy)(\lambda y.yyy)x.$
- (iv)  $\lambda xy.(\lambda z.xzz)\mathbf{\Omega}.$

**3.c** Which of the following reduction sequences are standard?

- (i)  $(\lambda x.x)((\lambda y.yy)(\lambda y.yy)) \rightarrow_{\beta} (\lambda x.x)((\lambda y.yy)(\lambda y.yy)) \rightarrow_{\beta} (\lambda y.yy)(\lambda y.yy).$
- (ii)  $(\lambda x.(\lambda y.(\lambda z.(\lambda pq.q)z)y)x)\mathbf{\Omega} \rightarrow_{\beta} (\lambda y.(\lambda z.(\lambda pq.q)z)y)\mathbf{\Omega}$   
 $\rightarrow_{\beta} (\lambda z.(\lambda pq.q)z)\mathbf{\Omega}$   
 $\rightarrow_{\beta} (\lambda pq.q)\mathbf{\Omega}$   
 $\rightarrow_{\beta} \lambda q.q$

**3.d** Find the standard reduction sequence from  $(\lambda x.xx)((\lambda y.yy)((\lambda z.zz)\mathbf{k}))$  to  $\mathbf{k}\mathbf{k}$ . Now find the shortest reduction sequence.

**3.e** Find terms  $t_1, \dots, t_n$  such that each of the following terms  $s$  satisfies  $\lambda\beta \vdash st_1 \dots t_n = \mathbf{i}$ :

- (i)  $\lambda xy.xxy.$
- (ii)  $(\lambda xy.yxxy)(\lambda xy.yxxy).$
- (iii)  $(\lambda xy.xy\mathbf{\Omega})(\lambda xy.xy\mathbf{\Omega}).$

[Hint: reduce the terms to hnf first.]

## Exercises

**3.1** Which of the following have  $\beta$ -normal forms, and which are solvable? For the former give the leftmost reduction sequence; for the latter the head reduction sequence.

- (i)  $\mathbf{sk}.$
- (ii)  $(\lambda x.xxx)(\lambda x.xxx).$
- (iii)  $\mathbf{yi}.$
- (iv)  $\mathbf{y}(\lambda yz.zy).$

**3.2** Let  $\omega \equiv \lambda x.xx$ . Draw the graph of all  $\beta$ -reductions from the term  $(\lambda y.\omega y)(\lambda y.\omega y)$  and indicate which of the reductions are leftmost.

**3.3** Show that any term which has a  $\beta$ -normal form is solvable.

**3.4** Let  $\omega \equiv \lambda ayz.z(aay)$  and  $s \equiv \omega\omega x$ .

- (i) Show that  $s$  has no  $\beta$ -normal form.  
 [Hint: Use Theorem 3.2.3.]

(ii) Show also that if  $s \twoheadrightarrow_{\beta} t$  then  $x \in FV(t)$ .

[Hint: consider a standard reduction of minimal length.]

The next 3 exercises go through the proof that the solvable terms coincide with the terms with a head normal form (at least for closed terms).

**3.5** Show that

(i) if  $s \xrightarrow{h} t$  then  $s[u/x] \xrightarrow{h} t[u/x]$ .

(ii) if  $st$  has a hnf then  $s$  has a hnf. Give an example to show that the converse is false.

**3.6** Use the Advancement of Head Reduction Lemma to show the following: if a  $\lambda\beta \vdash s = t$ , and  $s$  is in head normal form, then  $t$  has a hnf.

[Hint: first use the Church-Rosser property to show that there is a term  $u$ , itself in head normal form, which both  $s$  and  $t$  reduce to. Then consider  $t \twoheadrightarrow_{\beta} u$ .]

**3.7** Deduce that every solvable term has a hnf.

**3.8** Suppose that  $s$  has some infinite  $\beta$ -reduction sequence  $s \rightarrow_{\beta} s_1 \rightarrow_{\beta} s_2 \rightarrow_{\beta} \dots$  and that for infinitely many  $i$  it is true that  $s_i \xrightarrow{h} s_{i+1}$ . Use Lemma 3.1.1 to show that  $s$  is unsolvable.

Suppose that  $s$  has some infinite  $\beta$ -reduction sequence  $s \rightarrow_{\beta} s_1 \rightarrow_{\beta} s_2 \rightarrow_{\beta} \dots$  and that for infinitely many  $i$  it is true that  $s_i \xrightarrow{l} s_{i+1}$ . Is it the case that  $s$  must have no normal form?

**3.9** A closed term  $s$  is called **easy** if for all closed terms  $t$  the theory  $\lambda\beta + \{s = t\}$  is consistent (an example of an easy term is  $\Omega$ , although it is not particularly simple to show). Show that if  $s$  is easy then it is unsolvable.

**\*3.10** Let  $\mathbf{k}^{\infty} = \mathbf{y}\mathbf{k}$ . Show that  $\mathbf{k}^{\infty}$  is unsolvable. Hence show that  $s$  is solvable if and only if the theory  $\lambda\beta + \{s = \mathbf{k}^{\infty}\}$  is inconsistent.

**3.11** Use the Genericity Lemma to show that there is no term  $f$  such that  $f\mathbf{i}\Omega = \mathbf{k}$  and  $f\Omega\mathbf{i} = \mathbf{s}$ .

**3.12** Let  $p$  be any solvable term and  $q$  any unsolvable term.

(i) Show that  $\mathcal{H} + \{p = q\} = \lambda\beta + \{s = t \mid s, t \text{ are closed unsolvable terms}\} \cup \{p = q\}$  is inconsistent.

[Thus, any consistent sensible theory must be semi-sensible.]

- (ii) Let  $\mathcal{T}$  be a  $\lambda$ -theory with  $\mathcal{T} \vdash p = q$ . Show that either  $\mathcal{H}^* \vdash p = q$ , or  $\mathcal{T}$  is inconsistent.

[Thus, every consistent semi-sensible theory is contained in  $\mathcal{H}^*$ .]

Deduce that  $\mathcal{H}^*$  is the unique **Hilbert-Post complete** consistent sensible  $\lambda$ -theory. That is, for each equation  $s = t$  either  $\mathcal{H}^* \vdash s = t$  or  $\mathcal{H}^* + \{s = t\}$  is inconsistent.

**3.13** Show that  $\lambda\beta\eta \subseteq \mathcal{H}^*$ , so that the  $\eta$ -rule holds in  $\mathcal{H}^*$ . Show further that this inclusion is still proper.

Let  $\mathcal{H}\eta$  be the theory  $\lambda\beta\eta + \{s = t \mid s, t \text{ are closed unsolvable terms}\}$ . Show that  $\mathcal{H}\eta \subseteq \mathcal{H}^*$  and hence that  $\mathcal{H}\eta$  is consistent.

## Chapter 4

# Numerals and Recursive Functions

---

**Reading:** (*Using alternative numeral systems*)

[Bar84] 6.2–6.6, 8.4; [BB94] 3; [Han94] 6.2–6.4; [Sel07] 3.1–3.3

---

We now show that, although minimal in syntax, the  $\lambda$ -calculus is a very powerful language. But we need to talk about numbers, and numbers aren't part of the syntax. We show that numbers, and functions, can be encoded by terms and that as much power as one can expect from a programming language is available to compute with these numbers (principally because of the existence of fixed point combinators). We also prove the **Second Recursion Theorem**, which allows us to prove some undecidability results about the  $\lambda$ -calculus.

First, recall the combinators  $\mathbf{t} \equiv \lambda xy.x$  and  $\mathbf{f} \equiv \lambda xy.y$ . We will use  $\mathbf{t}$  to encode the boolean true and  $\mathbf{f}$  for false, because we can condition on the value of such a boolean very easily:  $\lambda\beta \vdash \mathbf{t}fg = f$  and  $\mathbf{f}fg = g$ .

### 4.1 Church Numerals

The traditional way to encode numbers in the untyped  $\lambda$ -calculus is due to Church. The number  $n$  is encoded by the term  $\ulcorner n \urcorner \equiv \lambda fx. \underbrace{f(f(f \dots (fx)))}_{n\text{-fold app}}$ .

**Definition** The **Church numerals**,  $\ulcorner n \urcorner$  for each  $n \in \mathbb{N}$ , and associated

combinators are given by:

$$\begin{aligned} \ulcorner 0 \urcorner &\equiv \lambda f x. x \quad (\equiv \mathbf{k}) \\ \ulcorner n + 1 \urcorner &\equiv \lambda f x. f(\ulcorner n \urcorner f x) \quad (\text{actually the normal form of this term is used}) \\ \mathbf{succ} &\equiv \lambda n f x. f(n f x) \\ \mathbf{zero?} &\equiv \lambda n. n(\lambda x. \mathbf{f}) \mathbf{t} \end{aligned}$$

The combinators have the following properties:

- (i)  $\mathbf{succ} \ulcorner n \urcorner = \ulcorner n + 1 \urcorner$ , so  $\mathbf{succ}$  is the successor function for Church numerals.
- (ii)  $\mathbf{zero?} \ulcorner 0 \urcorner = \mathbf{t}$ , and  $\mathbf{zero?} \ulcorner n + 1 \urcorner = \mathbf{f}$ . So  $\mathbf{zero?}$  is a test for zero.

Exercise 4.2 gives a combinator  $\mathbf{pred}$  with the property  $\mathbf{pred} \ulcorner n + 1 \urcorner = \ulcorner n \urcorner$ ,  $\mathbf{pred} \ulcorner 0 \urcorner = \ulcorner 0 \urcorner$ . So  $\mathbf{pred}$  is a (truncated) predecessor.

Another useful combinator is  $\mathbf{rcase} \equiv \lambda n f g. \mathbf{zero?} n f (g (\mathbf{pred} n))$  which satisfies

$$\mathbf{rcase} \ulcorner n \urcorner f g = \begin{cases} f, & \text{if } n = 0 \\ g(\ulcorner n - 1 \urcorner), & \text{if } n > 0. \end{cases}$$

## 4.2 Definability of Total Recursive Functions

Now that we can encode natural numbers in the  $\lambda$ -calculus, we can also encode functions on the natural numbers.

**Definition** A function  $\phi : \mathbb{N}^m \rightarrow \mathbb{N}$  is  $\lambda$ -**definable** by a term  $f$  if, for every tuple  $\langle n_1, \dots, n_m \rangle$  of natural numbers,

$$f \ulcorner n_1 \urcorner \dots \ulcorner n_m \urcorner = \ulcorner \phi(n_1, \dots, n_m) \urcorner.$$

We note that equality here, and indeed all other equalities of this chapter, can be replaced by  $\rightarrow_{\beta}^1$ . This is because of the Church-Rosser property of  $\beta$ -reduction and because all numerals and booleans are  $\beta$ -normal forms (via Lemma 2.4.1(ii)).

So we have already shown that the successor and predecessor unary functions<sup>2</sup> are  $\lambda$ -definable, by  $\mathbf{succ}$  and  $\mathbf{pred}$  respectively.

Recall from computability the definition of the total recursive functions:

**Definition** The set  $\mathcal{R}_0$  of **total recursive functions** is the set of functions  $\phi : \mathbb{N}^m \rightarrow \mathbb{N}$  for some  $m \in \mathbb{N}$ , given by the following rules:

<sup>1</sup>If so, in the proof of Theorem 4.2.1 the combinator  $\mathbf{y}$  should be replaced by  $\boldsymbol{\theta}$ , since  $\mathbf{y}$  does not have the property that  $\mathbf{y}f \rightarrow_{\beta} f(\mathbf{y}f)$  for all  $f$ ; see Exercise 2.4.

<sup>2</sup>The predecessor function we are talking about maps  $n + 1$  to  $n$  for all  $n$ , and 0 to 0.



- (Initial functions)

$$\text{(zero)} \quad \frac{}{\zeta : n \mapsto 0 \in \mathcal{R}_0}$$

$$\text{(successor)} \quad \frac{}{\sigma : n \mapsto n + 1 \in \mathcal{R}_0}$$

$$\text{(projections)} \quad \frac{}{\Pi_m^i : \langle n_1, \dots, n_m \rangle \mapsto n_i \in \mathcal{R}_0} \quad (1 \leq i \leq m)$$

- (Closure under Composition)

$$\frac{\chi : \mathbb{N}^m \rightarrow \mathbb{N} \in \mathcal{R}_0 \quad \psi_i : \mathbb{N}^l \rightarrow \mathbb{N} \in \mathcal{R}_0 \text{ for } 1 \leq i \leq m}{\phi : \langle n_1, \dots, n_l \rangle \mapsto \chi(\psi_1(\vec{n}), \dots, \psi_m(\vec{n})) \in \mathcal{R}_0}$$

- (Closure under Primitive Recursion)

$$\frac{\chi : \mathbb{N}^m \rightarrow \mathbb{N} \in \mathcal{R}_0 \quad \psi : \mathbb{N}^{m+2} \rightarrow \mathbb{N} \in \mathcal{R}_0}{\phi : \langle k, n_1, \dots, n_m \rangle \mapsto \begin{cases} \chi(n_1, \dots, n_m), & \text{if } k = 0 \\ \psi(\phi(k-1, n_1, \dots, n_m), k-1, n_1, \dots, n_m) & \text{if } k > 0 \end{cases} \in \mathcal{R}_0}$$

- (Closure under Minimalization)

$$\frac{\chi : \mathbb{N}^{m+1} \rightarrow \mathbb{N} \in \mathcal{R}_0}{\phi : \langle n_1, \dots, n_m \rangle \mapsto k, \text{ the least integer such that } \chi(k, n_1, \dots, n_m) = 0, \in \mathcal{R}_0}$$

In this last case there is a side condition on  $\chi$ , necessary for  $\phi$  to be a total function:

$$\forall m\text{-tuples } \langle n_1, \dots, n_m \rangle \exists k. \chi(k, n_1, \dots, n_m) = 0.$$

**Theorem 4.2.1 (Definability)** A function  $\phi : \mathbb{N}^m \rightarrow \mathbb{N}$  is recursive if and only if it is  $\lambda$ -definable.

**Proof** For  $(\Leftarrow)$  we appeal to Church's Thesis<sup>3</sup> (although it would be possible to prove directly it would be very tiresome). Let  $\phi : \mathbb{N}^m \rightarrow \mathbb{N}$  be defined by a term  $f$ . Then  $\phi(n_1, \dots, n_m)$  may be computed by the following procedure: take  $f \ulcorner n_1 \urcorner \dots \ulcorner n_m \urcorner$  and repeatedly apply leftmost one-step  $\beta$ -reduction until a normal form is reached. By the remark that equality in this chapter may be replaced by  $\rightarrow_\beta$ , and Theorem 3.2.3, this will terminate with

<sup>3</sup>We take Church's Thesis to mean: a total function is recursive if and only if it is computable by some algorithm (and similarly for partial functions).

the normal form  $\ulcorner \phi(n_1, \dots, n_m) \urcorner$ , from which we can immediately extract  $\phi(n_1, \dots, n_m)$ .

We show  $(\Rightarrow)$  by induction (equivalently, Lemma 1.1.1). We show that the initial functions are  $\lambda$ -definable, and that the  $\lambda$ -definable functions are closed under composition, primitive recursion, and minimalization.

- (Initial functions) It is easy to see that  $\sigma$  is defined by **succ**,  $\zeta$  by  $\mathbf{k} \ulcorner 0 \urcorner$ , and  $\Pi_m^i$  by  $\lambda x_1 \dots x_m. x_i$ .
- (Closure under Composition) Suppose that  $\chi$  is defined by  $f$ , and each  $\psi_i$  by  $g_i$ . Let  $h \equiv \lambda x_1 \dots x_l. f(g_1 \vec{x}) \dots (g_m \vec{x})$ . Then

$$\begin{aligned} h \ulcorner n_1 \urcorner \dots \ulcorner n_l \urcorner &= f(g_1 \ulcorner n_1 \urcorner \dots \ulcorner n_l \urcorner) \dots (g_m \ulcorner n_1 \urcorner \dots \ulcorner n_l \urcorner) \\ &= f \ulcorner \psi_1(n_1, \dots, n_l) \urcorner \dots \ulcorner \psi_m(n_1, \dots, n_l) \urcorner, \\ &\quad \text{because each } g_i \text{ defines } \psi_i \\ &= \ulcorner \chi(\psi_1(n_1, \dots, n_l) \dots \psi_m(n_1, \dots, n_l)) \urcorner, \\ &\quad \text{because } f \text{ defines } \chi \end{aligned}$$

hence  $h$  defines  $\phi : \langle n_1, \dots, n_l \rangle \mapsto \chi(\psi_1(\vec{n}), \dots, \psi_m(\vec{n}))$ .

- (Closure under Primitive Recursion) Suppose that  $\chi$  is defined by  $f$  and  $\psi$  by  $g$ . Let

$$h \equiv \mathbf{y}(\lambda s y x_1 \dots x_m. \mathbf{rcase} y (f \vec{x}) (\lambda z. g (s z \vec{x}) z \vec{x}))$$

Then  $h = \lambda y \vec{x}. \mathbf{rcase} y (f \vec{x}) (\lambda z. g (h z \vec{x}) z \vec{x})$ , so

$$\begin{aligned} h \ulcorner 0 \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_m \urcorner &= f \ulcorner n_1 \urcorner \dots \ulcorner n_m \urcorner \\ &= \ulcorner \chi(n_1, \dots, n_m) \urcorner, \text{ because } f \text{ defines } \chi. \end{aligned}$$

Also

$$\begin{aligned} h \ulcorner k + 1 \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_m \urcorner &= (\lambda z. g (h z \ulcorner n_1 \urcorner \dots \ulcorner n_m \urcorner) z \ulcorner n_1 \urcorner \dots \ulcorner n_m \urcorner) \ulcorner k \urcorner \\ &= g (h \ulcorner k \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_m \urcorner) \ulcorner k \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_m \urcorner. \end{aligned}$$

Now by an induction on  $k$  it follows that  $h$  defines  $\phi : \langle 0, n_1, \dots, n_m \rangle \mapsto \chi(n_1, \dots, n_m)$ ,  $\langle k+1, n_1, \dots, n_m \rangle \mapsto \psi(\phi(k, n_1, \dots, n_m), k, n_1, \dots, n_m)$ .

- (Closure under Minimalization) Suppose that  $\chi$  is defined by  $f$ . Let

$$g \equiv \mathbf{y}(\lambda s y x_1 \dots x_m. \mathbf{zero?} (f y \vec{x}) y (s (\mathbf{succ} y) \vec{x})).$$

Then  $g = \lambda y x_1 \dots x_m. \mathbf{zero?} (f y \vec{x}) y (g (\mathbf{succ} y) \vec{x})$ .

Because  $f$  defines  $\chi$ ,

$$g \ulcorner k \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_m \urcorner = \begin{cases} \ulcorner k \urcorner, & \text{if } \chi(k, n_1, \dots, n_m) = 0, \\ g \ulcorner k + 1 \urcorner \ulcorner n_1 \urcorner \dots \ulcorner n_m \urcorner, & \text{otherwise.} \end{cases}$$

So let  $h \equiv g \ulcorner 0 \urcorner$ . Exercise: show that  $h$  defines  $\phi : \langle n_1, \dots, n_m \rangle \mapsto$  the least integer such that  $\chi(k, n_1, \dots, n_m) = 0$ .

■

### 4.3 Undecidability Results

We can use Theorem 4.2.1 to prove more facts about the theory  $\lambda\beta$ . Given a language with enough power to compute recursive functions, a common technique is to encode the language *within itself*, and deduce that certain properties are undecidable.

**Definition** Note that there are countably many terms in  $\Lambda$ . It is easy to define an effective **Gödel numbering** for  $\Lambda$ , namely an effective (computable by some algorithm) bijection  $\sharp : \Lambda \rightarrow \mathbb{N}$ .

We also define  $\ulcorner - \urcorner : \Lambda \rightarrow \Lambda$  by  $\ulcorner s \urcorner = \ulcorner \sharp s \urcorner$ .

We can then prove the so-called Second Recursion Theorem:

**Theorem 4.3.1** For all terms  $f$  there is some term  $u$  such that  $f \ulcorner u \urcorner = u$ .

**Proof** By the effectiveness of the Gödel numbering function  $\sharp$ , there are recursive functions  $\text{app} : \mathbb{N}^2 \rightarrow \mathbb{N}$  and  $\text{gnum} : \mathbb{N} \rightarrow \mathbb{N}$  such that  $\text{app}(\sharp s, \sharp t) = \sharp(st)$  and  $\text{gnum}(n) = \sharp \ulcorner n \urcorner$ . These are therefore  $\lambda$ -definable, say by terms **app** and **gnum**, so

$$\text{app} \ulcorner s \urcorner \ulcorner t \urcorner = \ulcorner st \urcorner \text{ and } \text{gnum} \ulcorner s \urcorner = \ulcorner \ulcorner s \urcorner \urcorner$$

For each  $f$  define  $t \equiv \lambda x.f(\text{app } x(\text{gnum } x))$  and  $u \equiv t \ulcorner t \urcorner$ . Then

$$\begin{aligned} u &\equiv t \ulcorner t \urcorner \\ &= f \text{ app } \ulcorner t \urcorner (\text{gnum} \ulcorner t \urcorner) \\ &= f \text{ app } \ulcorner t \urcorner \ulcorner \ulcorner t \urcorner \urcorner \\ &= f \ulcorner t \ulcorner t \urcorner \urcorner \\ &\equiv f \ulcorner u \urcorner. \end{aligned}$$

■

Now we can show that there are no recursive functions which can decide equality in  $\lambda\beta^4$ .

**Theorem 4.3.2 (Scott-Curry)** Let  $\mathcal{A}$  and  $\mathcal{B}$  be two nonempty sets of terms which are closed under  $\beta$ -convertibility. There is no term  $f$  such that  $f \ulcorner s \urcorner = \mathbf{t}$  or  $\mathbf{f}$ , for all terms  $s$ , and

$$\lambda\beta \vdash f \ulcorner u \urcorner = \begin{cases} \mathbf{t}, & \text{if } u \in \mathcal{A} \\ \mathbf{f}, & \text{if } u \in \mathcal{B}. \end{cases}$$

**Proof** If  $\mathcal{A} \cap \mathcal{B} \neq \emptyset$  then the result is trivial, so assume that  $\mathcal{A}$  and  $\mathcal{B}$  are

<sup>4</sup>Compare the following theorem to Rice's Theorem from recursion theory.

disjoint.

Suppose that such an  $f$  exists. Let  $a \in \mathcal{A}$  and  $b \in \mathcal{B}$ . Define

$$g \equiv \lambda x. (fx)ba.$$

Then by the Second Recursion Theorem there is some  $u$  with

$$\lambda\beta \vdash u = g^{\ulcorner u \urcorner} = \begin{cases} b, & \text{if } f^{\ulcorner u \urcorner} = \mathbf{t} \\ a, & \text{if } f^{\ulcorner u \urcorner} = \mathbf{f}. \end{cases}$$

By assumption either  $f^{\ulcorner u \urcorner} = \mathbf{t}$  or  $f^{\ulcorner u \urcorner} = \mathbf{f}$ . If the former, we have  $u = b$  therefore  $u \in \mathcal{B}$ , therefore  $f^{\ulcorner u \urcorner} = \mathbf{t}$ , hence  $\mathbf{t} = \mathbf{f}$ , a contradiction. Similarly for the other case. ■

**Corollary 4.3.3** Let  $\mathcal{A}$  be a nonempty set not equal to the whole of  $\Lambda$  which is closed under  $\beta$ -convertibility. Then  $\mathcal{A}$  is not a recursive set.

**Proof** Take  $\mathcal{B} = \Lambda \setminus \mathcal{A}$  in Theorem 4.3.2, and use that fact that every total recursive function is definable in the  $\lambda$ -calculus. ■

## 4.4 Extension to Partial Functions

Computability is rooted in the study of **partial functions** (i.e. functions which are not necessarily defined at every point of their domain). Definability in the  $\lambda$ -calculus can be extended to partial functions in a mostly straightforward way. First, some notation.

**Definition** Let  $\phi : \mathbb{N}^m \rightarrow \mathbb{N}$  be a partial function. We write  $\phi(n_1, \dots, n_m) = k$  to mean that  $\phi$  is defined on  $\langle n_1, \dots, n_m \rangle$  and equal to  $k$ . We write  $\phi(n_1, \dots, n_m) \downarrow$  to mean that  $\phi$  is defined on  $\langle n_1, \dots, n_m \rangle$ . We write  $\phi(n_1, \dots, n_m) \uparrow$  to mean that  $\phi$  is not defined on  $\langle n_1, \dots, n_m \rangle$ .

We have to decide which term(s) will represent “undefined” numbers. For a start, it is impossible to find a single term (e.g.  $\Omega$ ) to do so in a consistent way, if we want the composition of functions to be defined by a composition of terms. Similarly, we cannot use the set of all terms without a  $\beta$ -normal form. In view of the discussion in Section 3.3 the natural choice is the set of unsolvable terms (and this suggests that the paradigm of computation as reduction should be, more precisely, computation as head reduction).

We use unsolvable terms to encode undefined into the  $\lambda$ -calculus as follows:

**Definition** A function  $\phi : \mathbb{N}^m \rightarrow N$  is **strongly  $\lambda$ -definable** by a term  $f$  if, for every tuple  $\langle n_1, \dots, n_m \rangle$  of natural numbers,

$$f^{\ulcorner n_1 \urcorner \dots \urcorner n_m \urcorner} \begin{cases} = \ulcorner \phi(n_1, \dots, n_m) \urcorner, & \text{if } \phi(n_1, \dots, n_m) \downarrow \\ \text{is unsolvable,} & \text{if } \phi(n_1, \dots, n_m) \uparrow \end{cases}$$

(= can be replaced by  $\xrightarrow{\text{h}}$  and unsolvable by “has no hnf”.) Recall the **partial recursive functions**, defined analogously to the total recursive functions except that:

- (i) all functions are allowed to be partial,
- (ii) the composition of partial functions is defined at a certain tuple only if **all** the composed functions are defined at that point (even if those functions are not actually used in the composition),
- (iii) there is no constraint on closure under minimalization. The result is defined only if there is some  $k$  such that  $\chi(k', n_1, \dots, n_m)$  is defined and not equal to zero for all  $k' < k$ , and  $\chi(k, n_1, \dots, n_m) = 0$ .

Then we have the result known as **Turing Completeness**:

**Theorem 4.4.1** A partial function  $\phi : \mathbb{N}^m \rightarrow \mathbb{N}$  is partial recursive if and only if it is strongly  $\lambda$ -definable.

**Proof** We will not include the proof, which is similar to the total case. See [Bar84, §8.4]. Item (ii), above, causes some complications because we are forced to perform composition as a two-stage process: first testing that all composed functions are defined, and then performing the composition. Inherently, the  $\lambda$ -calculus (under head reduction) has non-strict behaviour, whereas composition of partial recursive functions is supposed to be strict. ■

The moral of this result is that the  $\lambda$ -calculus is every bit as powerful — at least as far as computing functions goes — as any other computing language or device. This is not to say that computing functions with numbers encoded as numerals is convenient!

## Computational Practice

- 4.a** Which numeral happens to be the normal form of  $\ulcorner 2 \urcorner \ulcorner 2 \urcorner$ ?
- 4.b** Which function is defined by the term  $\lambda n f x. \mathbf{zero} ? n x (fx)$ ?
- 4.c** Which function is defined by the term  $\lambda n f x. n(\lambda x. f(fx))$ ?

**4.d** Find a term **not** which satisfies **not t = f** and **not f = t**.

---

## Exercises

**4.1** Find a term **and** which satisfies all of:

- (i) **and t t = t**,
- (ii) **and t f = f**,
- (iii) **and f t = f**,
- (iv) **and f f = f**.

Does your term give the same “answer” for **and Ω f** and **and f Ω**? If **Ω** represents undefined, what “should” be the result of these two computations?

**4.2** Show that

$$\mathbf{pred}' \equiv \lambda n.n(\lambda y.y \mathbf{i} (\mathbf{succ} y))(\lambda ab.\ulcorner 0 \urcorner)$$

works as a predecessor for the Church numerals, except that **pred' ∖0∖ ≠ ∖0∖** (you will need to use induction).

Explain how to modify **pred'** to **pred** so that **pred** has the same properties except that **pred ∖0∖ = ∖0∖**.

**4.3** Complete the details of the case of Closure under Minimalization in the proof of Theorem 4.2.1.

**4.4** Give explicitly a  $\lambda$ -term which defines the function  $eq : \mathbb{N}^2 \rightarrow \mathbb{N}$  where

$$eq(m, n) = \begin{cases} 1, & \text{if } m = n, \\ 0, & \text{if } m \neq n. \end{cases}$$

[Hint: Do not use Theorem 4.2.1. Instead, use **zero?** to make a term  $f$  such that

$$f \ulcorner m \urcorner \ulcorner n \urcorner = \begin{cases} \ulcorner 1 \urcorner, & \text{if } m = n = 0, \\ \ulcorner 0 \urcorner, & \text{if } m = 0 \text{ and } n \neq 0 \text{ or } n = 0 \text{ and } m \neq 0, \\ f \ulcorner m - 1 \urcorner \ulcorner n - 1 \urcorner, & \text{otherwise.} \end{cases}$$

Then show that  $f$  defines  $eq$ .]

**4.5** We can specify an alternative choice of numerals by a sequence of closed terms  $\mathbf{d} = \langle \mathbf{d}_0, \mathbf{d}_1, \dots \rangle$ . This is called a **numeral system** if there are terms  $\mathbf{succ}_{\mathbf{d}}$  and  $\mathbf{zero?}_{\mathbf{d}}$  such that

$$\mathbf{succ}_{\mathbf{d}} \mathbf{d}_n = \mathbf{d}_{n+1}, \quad \mathbf{zero?}_{\mathbf{d}} \mathbf{d}_0 = \mathbf{t}, \quad \text{and} \quad \mathbf{zero?}_{\mathbf{d}} \mathbf{d}_{n+1} = \mathbf{f}.$$

Suppose that for a numeral system  $\mathbf{d}$  there is also a term  $\mathbf{pred}_{\mathbf{d}}$  with

$$\mathbf{pred}_{\mathbf{d}} \mathbf{d}_{n+1} = \mathbf{d}_n \text{ for each } n \in \mathbb{N}.$$

Show that there are terms  $f$  and  $g$  such that  $f\mathbf{d}_n = \ulcorner n \urcorner$  and  $g\ulcorner n \urcorner = \mathbf{d}_n$  for all  $n \in \mathbb{N}$ .

**4.6** Say that a function  $\phi : \mathbb{N}^m \rightarrow \mathbb{N}$  is  $\lambda$ -definable with respect to a numeral system  $\mathbf{d}$  if there is a term  $f$  such that for every tuple  $\langle n_1, \dots, n_m \rangle$  of natural numbers,

$$f \mathbf{d}_{n_1} \dots \mathbf{d}_{n_m} = \mathbf{d}_{\phi(n_1, \dots, n_m)}.$$

Suppose that  $\mathbf{d}$  has a predecessor term as in the previous exercise. Use the previous exercise and Theorem 4.2.1 to show that every recursive function is  $\lambda$ -definable with respect to  $\mathbf{d}$ .

**\*4.7** Give an effective Gödel numbering of terms, namely an effectively computable (i.e. given by some algorithm) bijection between  $\Lambda$  and  $\mathbb{N}$ .

Remember that terms which are  $\alpha$ -convertible are considered to be the same (from Chapter 1, the set of terms is really  $\Lambda$  quotiented by  $\equiv$ ) so your encoding should be invariant on  $\alpha$ -convertible terms.

[Hint: Either give a way to rename canonically all bound variables in a term, so that all  $\alpha$ -convertible terms end up renamed to the same thing, or look up and use **de Bruijn notation** for bound variables.]

**4.8** Prove the following modification of the Scott-Curry Theorem:

Let  $\mathcal{A}$  and  $\mathcal{B}$  be non-empty sets of terms which are closed under  $\beta$ -convertibility. Show that  $\mathcal{A}$  and  $\mathcal{B}$  are **recursively inseparable**, i.e. there is no recursive set<sup>5</sup>  $\mathcal{C}$  with  $\mathcal{A} \subseteq \mathcal{C}$  and  $\mathcal{B} \cap \mathcal{C} = \emptyset$ .

**4.9** Let  $\mathcal{A} = \{s \mid s \text{ has a normal form}\}$ . Show that  $\mathcal{A}$  is a recursively enumerable<sup>6</sup> set which is not recursive.

<sup>5</sup>A recursive set  $S$  is one for which the **characteristic function**  $\chi_S : s \in S \mapsto 1, s \notin S \mapsto 0$  is recursive.

<sup>6</sup>A recursively enumerable, also called r.e. or semi-decidable, set is one for which there is a partial recursive function  $\chi_S$  for which  $\chi_S(s) = 1$  if and only if  $s \in S$ .  $\chi_S$  may be undefined on elements not in  $S$ .

**4.10** To say that a theory  $\mathcal{T}$  is decidable means that there is a recursive function  $\phi : \mathbb{N}^2 \rightarrow \mathbb{N}$  such that

$$\phi(\#s, \#t) = \begin{cases} 1, & \text{if } \mathcal{T} \vdash s = t \\ 0, & \text{otherwise.} \end{cases}$$

Show that  $\lambda\beta$  is not decidable.

**4.11**

- (i) For each term  $f$  use the First Recursion Theorem to find a term  $g_f$  with  $g_f = \lambda xyz.g_f f z x$ . Show that  $g_f s g_f t = g_f f t s$ .
- (ii) Let  $\phi : \mathbb{N}^2 \rightarrow \mathbb{N}$  be recursive. Construct distinct closed terms  $s_0, s_1, \dots$  such that for all  $m, n \in \mathbb{N}$ ,

$$s_m s_n = s_{\phi(m,n)}.$$

[Hint: Consider  $s_n \equiv \lambda x.xh\ulcorner n \urcorner$  for an appropriate term  $h$ .]

**\*4.12** Show that the requirement that  $\phi$  be recursive cannot be omitted in (ii) above.

**4.13** Give explicitly a  $\lambda$ -term which defines the partial function  $\phi : \mathbb{N} \rightarrow \mathbb{N}$  where

$$\begin{cases} \phi(n) = n + 1, & \text{if } n \text{ is even} \\ \phi(n)\uparrow, & \text{if } n \text{ is odd.} \end{cases}$$

[Hint: First check the parity of  $n$ , and only afterwards compute  $n + 1$ .]

**4.14** Let  $\mathbf{test} \equiv \lambda n.n\mathbf{ii}$ . Show that, for any numeral  $\ulcorner n \urcorner$ ,  $\mathbf{test} \ulcorner n \urcorner = \mathbf{i}$ , but that for any unsolvable term  $s$ ,  $\mathbf{test} s$  is unsolvable.

(This term is used in the extension of Theorem 4.2.1 to partial functions, to test whether something is defined but discarding the numerical result if it is defined.)

**4.15** To say that the property of solvability is recursive means that there is a recursive function  $\phi : \mathbb{N} \rightarrow \mathbb{N}$  such that

$$\phi(\#s) = \begin{cases} 1, & \text{if } s \text{ is solvable} \\ 0, & \text{if } s \text{ is unsolvable.} \end{cases}$$

for all terms  $s$ .

Show that solvability is not recursive, but give an algorithm which does correctly say when a term is solvable (the algorithm may fail to terminate when the term is not solvable).



# Chapter 5

## Models

---

**Reading:** [Bar84] 2.2, Ch. 7, 18.1; [Han94] Ch. 4, 5.1;  
[Sel07] Ch. 5

---

We give a brief introduction to the model theory of the untyped  $\lambda$ -calculus. Rather than give a direct denotational semantics to terms, we take a small detour (traditional and interesting, not to mention easier) through **combinatory logic**. Combinatory logic was developed in parallel with the  $\lambda$ -calculus and with somewhat similar aims, and there are close connections: terms of the untyped  $\lambda$ -calculus can be translated to terms of combinatory logic and back, although the correspondence is not totally faithful and this mismatch is quite interesting.

Models of combinatory logic are called **combinatory algebras**; translating  $\lambda$ -terms to combinatory logic terms, we now have a definition for a model of the untyped  $\lambda$ -calculus. By a “model” we mean some function  $\llbracket - \rrbracket$  which translates terms of the  $\lambda$ -calculus into elements of some set, and with the property that  $\llbracket st \rrbracket$  is some natural function of  $\llbracket s \rrbracket$  and  $\llbracket t \rrbracket$ ; because all terms in the  $\lambda$ -calculus are also, in some sense, functions (by application) it is difficult to find any set with this property. We also want that  $\llbracket s \rrbracket = \llbracket t \rrbracket$  when  $s$  and  $t$  are equal in  $\lambda\beta$ .

Combinatory algebras are interesting in their own right, and have some unusual properties. A classic result is that they are **combinatory complete**. Finally, we will present a simple combinatory algebra and examine it as a model of the  $\lambda$ -calculus.

### 5.1 Combinatory Logic

We will abbreviate the phrase Combinatory Logic to CL. As with the  $\lambda$ -calculus we begin with a set of terms.

**Definition** The set of **CL-terms**,  $\mathcal{T}_{CL}$ , is given by the rules

$$\frac{}{x \in \mathcal{T}_{CL}} \quad x \in \mathcal{V} \qquad \frac{A \in \mathcal{T}_{CL} \quad B \in \mathcal{T}_{CL}}{(A \cdot B) \in \mathcal{T}_{CL}}$$

$$\frac{}{\mathbf{K} \in \mathcal{T}_{CL}} \qquad \frac{}{\mathbf{S} \in \mathcal{T}_{CL}}$$

where  $\mathbf{K}$  and  $\mathbf{S}$  are distinguished symbols or constants.

In general we will use capital roman letters for terms of CL, to distinguish them from terms of the  $\lambda$ -calculus. However, we may as well allow the set of object variables of the two languages to coincide, so we continue to use lowercase roman letters for variables in CL. If  $A$  and  $B$  are terms of CL we tend to omit the  $\cdot$  and write  $AB$  for  $A \cdot B$ , and  $ABC \dots Z$  for  $((AB)C) \dots Z$ . We write  $A[B/x]$  for the term which is obtained by replacing every (free) occurrence of  $x$  in  $A$  with  $B$ . (Since there are no variable binders, every variable occurrence is free in each term.) We use the notation  $A \equiv B$  to mean that  $A$  and  $B$  are identical terms in CL.

Note that there is no  $\alpha$ -conversion, because there are no free variables. This could be viewed as a definite advantage.

**Definition** Equality between CL terms is defined inductively according to the rules:

$$\begin{array}{ll} \text{(reflexivity)} & \frac{}{A = A} \\ \text{(symmetry)} & \frac{A = B}{B = A} \\ \text{(transitivity)} & \frac{A = B \quad B = C}{A = C} \\ \text{(application)} & \frac{A = A' \quad B = B'}{AB = A'B'} \\ \text{(K)} & \frac{}{\mathbf{K}AB = A} \\ \text{(S)} & \frac{}{\mathbf{S}ABC = AC(BC)} \end{array}$$

We write  $CL \vdash A = B$  when  $A = B$  is provable in this system.

The constants  $\mathbf{K}$  and  $\mathbf{S}$  behave like the terms (combinators!)  $\mathbf{k}$  and  $\mathbf{s}$  in the  $\lambda$ -calculus. A simple example:  $CL \vdash \mathbf{SKKA} = \mathbf{KA}(\mathbf{KA}) = A$  for any CL-term  $A$ . Conventionally, we write  $\mathbf{I}$  for  $\mathbf{SKK}$ .

## 5.2 Simulating Abstraction in Combinatory Logic

There is no abstraction in CL, but it can be simulated by the following **abstraction algorithm**.

**Definition** The operation **abstraction** is a family of maps  $abs_x : \mathcal{T}_{CL} \rightarrow \mathcal{T}_{CL}$ , one for each variable  $x$ . The term  $abs_x(A)$  is conventionally written as  $\lambda x.A$  and is defined by recursion:

$$\begin{aligned} \lambda x.x &\equiv \mathbf{SKK} \\ \lambda x.A &\equiv \mathbf{KA}, && \text{if } x \text{ does not occur in } A \\ \lambda x.AB &\equiv \mathbf{S}(\lambda x.A)(\lambda x.B), && \text{if } x \text{ does occur in } AB \end{aligned}$$

We also write  $\lambda x_1 \dots x_n.A$  for  $\lambda x_1.(\dots(\lambda x_n.A))$ .

Do not confuse the symbols  $\lambda$  and  $\lambda$ . The first makes up the grammar of the  $\lambda$ -calculus — it is part of that language — and the second is an **operation** on terms of combinatory logic. Nonetheless, they have similar properties, as the following so-called “ $\beta$ -simulation” lemma shows.

### Lemma 5.2.1 ( $\beta$ -simulation)

- (i)  $x$  does not occur free in  $\lambda x.A$  (i.e. it does not occur at all),
- (ii) For all  $B$ ,  $CL \vdash (\lambda x.A)B = A[B/x]$ ,
- (iii) Similarly  $CL \vdash (\lambda x_1 \dots x_n.A)B_1 \dots B_n = A[B_1/x_1] \dots [B_n/x_n]$ .
- (iv) If  $x$  is distinct from  $y$  and does not appear in  $B$  then  $(\lambda x.A)[B/y] \equiv \lambda x.(A[B/y])$ .

### Proof

- (i) Is immediate from the definition.
- (ii) By induction on  $A$ . If  $A \equiv x$  then  $CL \vdash (\lambda x.A)B \equiv \mathbf{SKKB} = B \equiv A[B/x]$ . If  $x$  does not occur in  $A$  then  $CL \vdash (\lambda x.A)x \equiv \mathbf{KAB} = A \equiv A[B/x]$ . If  $A \equiv CD$  and  $x$  occurs in  $CD$  then  $CL \vdash (\lambda x.CD)B \equiv \mathbf{S}(\lambda x.C)(\lambda x.D)B = (\lambda x.C)B((\lambda x.D)B) = C[B/x]D[B/x]$ , by the induction hypothesis, and this is  $A[B/x]$ .
- (iii) Inductively from (ii).
- (iv) Another induction on  $A$ . ■

### 5.3 Translations to and from the Lambda Calculus

Combinatory logic and the  $\lambda$ -calculus are closely related: there are strong connections between their formal theories. There are also natural translations between the two sets of terms.

**Definition** Assume that the variables of combinatory logic and the  $\lambda$ -calculus coincide. We define two maps  $(-)_cl : \Lambda \rightarrow \mathcal{T}_{CL}$  and  $(-)_\lambda : \mathcal{T}_{CL} \rightarrow \Lambda$  inductively by:

$$\begin{array}{ll} x_{cl} \equiv x & x_\lambda \equiv x \\ (st)_{cl} \equiv s_{cl}t_{cl} & (AB)_\lambda \equiv A_\lambda B_\lambda \\ (\lambda x.s)_{cl} \equiv \mathbb{K}x.(s_{cl}) & \mathbf{K}_\lambda \equiv \mathbf{k} \\ & \mathbf{S}_\lambda \equiv \mathbf{s} \end{array}$$

The translation is of interest for the practical implementation of (variations of) the  $\lambda$ -calculus on a computer. Translating terms into CL is like “compiling” them into a very small abstract language (consisting only of  $\mathbf{K}$  and  $\mathbf{S}$ , and possibly some variables). The “computation” on CL terms is natural and simple (see below for details of reduction in CL) because there are no substitutions to perform. On the other hand, the “compiled” CL term might be very much larger than the original  $\lambda$ -term, so in practice a richer structure with some additional constants is used. The implementation of functional languages is a fascinating subject, and the theory of the  $\lambda$ -calculus and combinatory logic has done much to inform it.

The faithfulness of the correspondence between  $\lambda\beta$  and  $CL$  is summed up by the following:

**Lemma 5.3.1**

- (i)  $\lambda\beta \vdash (\mathbb{K}x.A)_\lambda = \lambda x.A_\lambda$ .
- (ii) Hence  $\lambda\beta \vdash (s_{cl})_\lambda = s$  for all terms  $s$  (in fact  $(s_{cl})_\lambda \rightarrow_\beta s$ ).
- (iii) Also  $CL \vdash A = B \implies \lambda\beta \vdash A_\lambda = B_\lambda$ .

**Proof** Three straightforward inductions. Exercises. ■

But that is about as far as the correspondence goes. For example,  $(\mathbf{K}_\lambda)_{cl} = (\lambda xy.x)_{cl} = \mathbb{K}xy.x$  which is not provably equal to  $\mathbf{K}$  in  $CL$  (can you see why?). Also  $\lambda\beta \vdash \mathbf{sk} = \mathbf{k}(\mathbf{skk})$ , but  $CL \not\vdash \mathbf{SK} = \mathbf{K}(\mathbf{SKK})$  (again, can you see why?).

The main difference is that the equivalent rule to

$$\frac{s = t}{\lambda x.s = \lambda x.t}$$

is not valid in  $CL$ . For example  $CL \vdash \mathbf{I}x = x$  but  $CL \not\vdash \lambda x.Ix = \lambda x.x$  (again, why is this?). There are not “enough” equalities in  $CL$  to match the equalities of  $\lambda\beta$ . We will need some additional rules for  $CL$  to get the perfect correspondence of equality:

- (i)  $\lambda\beta \vdash (s_{cl})_\lambda = s$ .
- (ii)  $CL + \text{additional rules} \vdash (A_\lambda)_{cl} = A$ .
- (iii)  $\lambda\beta \vdash s = t$  if and only if  $CL + \text{additional rules} \vdash s_{cl} = t_{cl}$ .
- (iv)  $CL + \text{additional rules} \vdash A = B$  if and only if  $\lambda\beta \vdash A_\lambda = B_\lambda$ .

Traditionally there are three ways to strengthen equality in combinatory logic, but the details of why they work are (just) beyond the syllabus and we will leave all the details for optional exercises.

- (i) Add the **extensionality rule**:

$$(\text{Ext}) \frac{Ax = Bx}{A = B} \quad (x \text{ does not occur in } A \text{ or } B)$$

Although a simple and intuitive rule, this adds “too many” equalities. In Exercises 5.7 and 5.8 show that there is in fact a tight correspondence between  $CL + \text{Ext}$  and  $\lambda\beta\eta$ .

- (ii) Add the **weak extensionality rule**:

$$(\text{WExt}) \frac{A = B}{\lambda x.A = \lambda x.B}$$

for each variable  $x$ . Perhaps surprisingly this is not quite enough by itself. We also need to add

$$(\text{K-Ext}) \frac{}{\mathbf{K} = \lambda xy.\mathbf{K}xy} \quad (\text{S-Ext}) \frac{}{\mathbf{S} = \lambda xyz.\mathbf{S}xyz}$$

Exercises 5.5 and 5.6 shows that the equalities in  $\lambda\beta$  and  $CL + \text{WExt} + \text{K-Ext} + \text{S-Ext}$  do match perfectly. (Note that some authors use “CL” to mean what we have called  $CL$  along with these extra rules  $\text{WExt} + \text{K-Ext} + \text{S-Ext}$ .)

- (iii) One does not need to use a formula such as  $\text{WExt}$ . In [CF58] Curry showed that 5 axioms (rules with no premises) suffice; they are known as  $A_\beta$ :

- (A1)  $\mathbf{K} = \mathbf{S}(\mathbf{S}(\mathbf{K}\mathbf{S})(\mathbf{S}(\mathbf{K}\mathbf{K})\mathbf{K}))(\mathbf{K}(\mathbf{S}\mathbf{K}\mathbf{K}))$ ,
- (A2)  $\mathbf{S} = \mathbf{S}(\mathbf{S}(\mathbf{K}\mathbf{S})(\mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{K}\mathbf{S})))\mathbf{S}))(\mathbf{K}(\mathbf{K}(\mathbf{S}\mathbf{K}\mathbf{K})))$ ,
- (A3)  $\mathbf{S}(\mathbf{S}(\mathbf{K}\mathbf{S})(\mathbf{S}(\mathbf{K}\mathbf{K})(\mathbf{S}(\mathbf{K}\mathbf{S})\mathbf{K})))(\mathbf{K}\mathbf{K}) = \mathbf{S}(\mathbf{K}\mathbf{K})$ ,
- (A4)  $\mathbf{S}(\mathbf{K}\mathbf{S})(\mathbf{S}(\mathbf{K}\mathbf{K})) = \mathbf{S}(\mathbf{K}\mathbf{K})(\mathbf{S}(\mathbf{S}(\mathbf{K}\mathbf{S})(\mathbf{S}(\mathbf{K}\mathbf{K})(\mathbf{S}\mathbf{K}\mathbf{K})))\mathbf{K}(\mathbf{S}\mathbf{K}\mathbf{K}))$ ,
- (A5)  $\mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{K}\mathbf{S})))\mathbf{S}(\mathbf{K}\mathbf{S})(\mathbf{S}(\mathbf{K}\mathbf{S}))$   
 $= \mathbf{S}(\mathbf{S}(\mathbf{K}\mathbf{S})(\mathbf{S}(\mathbf{K}\mathbf{K})(\mathbf{S}(\mathbf{K}\mathbf{S})(\mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{K}\mathbf{S})))\mathbf{S})))\mathbf{K}\mathbf{S}$ .

(A1 and A2 are (K-Ext) and (S-Ext)). You can follow their construction in [Bar84, §7.3]. The existence of a finite set of such axioms is important for the model theory of the  $\lambda$ -calculus.

## 5.4 More Analogies Between CL and $\lambda$ -calculus

The similarities between  $CL$  (without extra rules) and  $\lambda$ -calculus are deep, but there are also some very interesting mismatches (of which the equality mismatch is only the surface). Here is a brief summary:

- There is a natural notion of reduction on  $\mathcal{T}_{CL}$ : all instances of  $\mathbf{K}AB$  reduce to  $A$  and all instances of  $\mathbf{S}ABC$  reduce to  $AC(BC)$ . The compatible closure is called **weak reduction**, written  $\rightarrow_w$  (there is also a **strong reduction** but it is rather complicated and not so interesting). Weak reduction does not match  $\beta$ -reduction – it is strictly weaker, with

$$A \rightarrow_w B \implies A_\lambda \rightarrow_\beta B_\lambda$$

but not vice versa (e.g.  $\mathbf{SK}$  is in  $w$ -normal form but the corresponding  $\lambda$ -term has a redex).

- Weak reduction has the Church-Rosser property. One proof is to introduce a new notion of **parallel weak reduction**:

$$\frac{}{A \rightsquigarrow A} \quad \frac{}{\mathbf{K}AB \rightsquigarrow A} \quad \frac{}{\mathbf{S}ABC \rightsquigarrow AC(BC)}$$

$$\frac{A \rightsquigarrow A' \quad A \rightsquigarrow B'}{AB \rightsquigarrow A'B'}$$

and show that it has the diamond property, and that its transitive closure is  $\rightarrow_w$ .

Therefore weak reduction has unique normal forms.

- The mismatches between weak reduction in  $CL$  and  $\beta$ -reduction in the  $\lambda$ -calculus are particularly interesting, especially as  $CL$  is the more natural language to implement on a computer. It suggests that full  $\beta$ -reduction is actually too strong, and that some  $\beta$ -reductions (e.g. those inside abstractions) should not be allowed. We have already noted that head reduction corresponds better with implementation of functional programming, and these further mismatches suggest that we should use **weak head reduction**, which does not allow reduction either inside an abstraction or on the right of an application.

One example of the reduction mismatch: in  $CL$ , if a term has one cycle of  $w$ -reductions, then it must have infinitely many different cycles.

This is not so for  $\beta$ -reduction (e.g.  $\Omega$ ). (See [Bar84, §7.3] for some other examples.)

- It is possible to encode numerals in CL. All (partial) recursive functions are definable. There is a “second recursion theorem”, and equality of terms in CL is therefore undecidable.
- (Looking forward to the next few chapters.) There are natural simple type system for CL; results analogous to those for simple types in  $\lambda\beta$ , including strong normalization and poor definability power, still hold.

## 5.5 Combinatory Algebras

A combinatory algebra is a model of combinatory logic. More precisely,

**Definition** A **combinatory algebra** has four parts: a set  $\mathcal{A}$ , a binary operation  $\bullet$  on  $\mathcal{A}$ , and two elements  $K, S \in \mathcal{A}$ , such that

$$\begin{aligned} \forall X, Y \in \mathcal{A}. \quad K \bullet X \bullet Y &= X \\ \forall X, Y, Z \in \mathcal{A}. \quad S \bullet X \bullet Y \bullet Z &= X \bullet Z \bullet (Y \bullet Z) \end{aligned}$$

(Conventionally we assume that  $\bullet$  associates to the left, and we will elide the symbol altogether more often than not.)

**Note for logicians:** When combinatory logic is formulated as a first-order language with equality, this is precisely a “model” of that language, in the sense of Tarski.

Given a combinatory algebra  $\mathcal{A}$  we can translate terms of CL into elements of  $\mathcal{A}$  in a systematic way:  $\mathbf{K}$  to  $K$ ,  $\mathbf{S}$  to  $S$ , and  $A \cdot B$  to  $A \bullet B$ . This works for *closed* terms. Terms with free variables are in some sense “incomplete”, and could be thought of as translating to functions on  $\mathcal{A}$ . For example, the term  $SKx$  translates to  $(S \bullet K) \bullet x$ , or if you like to the function which maps  $a \in \mathcal{A}$  to  $(S \bullet K) \bullet a$ . We will use the following formal definition:

**Definition** Given a combinatory algebra  $\mathcal{A}$ , an **environment** (or **valuation**) is a map  $\rho : \mathcal{V} \rightarrow \mathcal{A}$ . In an environment  $\rho$  the **denotation** of a CL-term  $A$  is written  $(A)_\rho$  and is given inductively by:

$$\begin{aligned} (x)_\rho &= \rho(x) \\ (\mathbf{K})_\rho &= K \\ (\mathbf{S})_\rho &= S \\ (A \cdot B)_\rho &= (A)_\rho \bullet (B)_\rho \end{aligned}$$

In this definition, the environment specifies what to “do” with free variables. When  $A$  has no free variables, the environment is irrelevant and we can just write  $(A)$ . Because the CL-terms  $\mathbf{K}$  and  $\mathbf{S}$  satisfy the same properties (w.r.t. “.”) as the elements  $K$  and  $S$  of combinatory algebras (w.r.t. “ $\bullet$ ”) it is trivial to note that every equation provable in  $CL$  remains true when interpreted in any combinatory algebra. The connection works both ways in this sense:

**Lemma 5.5.1**  $CL \vdash A = B$  if and only if:

in *every* combinatory algebra  $\mathcal{A}$  (and every environment  $\rho$ ) we have  $(A)_\rho = (B)_\rho$ .

Note that *some* combinatory algebras might have  $(A)_\rho = (B)_\rho$ , for all  $\rho$ , even when  $CL \not\vdash A = B$  (e.g. the trivial one-element combinatory algebra).

The map from combinatory logic allows us to use combinatory algebras, indirectly, as models for the  $\lambda$ -calculus.

**Definition** Let  $\mathcal{A}$  be a combinatory algebra, and  $\rho$  an environment. The **denotation** of a term  $s \in \Lambda$  (in the environment  $\rho$ ) is the element of  $\mathcal{A}$  given by

$$\llbracket s \rrbracket_\rho = (s_{cl})_\rho.$$

Because of the connection between equality in  $\lambda\beta$  and equality in combinatory logic, combinatory algebras make for good models of the  $\lambda$ -calculus. However, we have already seen that equality in  $\lambda\beta$  and equality in combinatory logic do not match exactly:  $CL$  equates fewer terms than  $\lambda\beta$ . Also, combinatory algebras might equate other things equated in neither  $CL$  nor  $\lambda\beta$ . The latter does not bother us (and seems to be unavoidable, in fact), but it is common to consider only combinatory algebras with the following property:

**Definition** If a combinatory algebra has the property that  $\lambda\beta \vdash s = t \implies \llbracket s \rrbracket_\rho = \llbracket t \rrbracket_\rho$  for every environment  $\rho$ <sup>1</sup> then we call it a  **$\lambda$ -algebra**.

(Note: strictly speaking, a  $\lambda$ -algebra must also have the property that  $\llbracket \lambda xy.x \rrbracket = K$  and  $\llbracket \lambda xyz.xz(yz) \rrbracket = S$ .) For more on models of the  $\lambda$ -calculus, see [Bar84, Ch. 5]. There is a very general description of  $\lambda$ -algebras in terms of cartesian closed categories, but that is beyond the scope of this course.

Combinatory algebras have very simple axioms which look rather innocuous. Actually, they are what Barendregt calls “algebraically pathological”, because of the following properties:

<sup>1</sup>Remember that if  $s$  and  $t$  are closed terms then the environment is irrelevant anyway.



**Lemma 5.5.2** Combinatory algebras (except the trivial one, which has only one element) are

- (i) never commutative,
- (ii) never associative,
- (iii) never finite.

**Proof** We elide the  $\bullet$  operation, and write  $I$  for  $SKK$ .

- (i) Suppose  $IK = KI$ . Then  $K = IK = KI$  so  $A = KAB = KIAB = IB = B$  for all  $A$  and  $B$  so the combinatory algebra is trivial.
- (ii) Suppose  $(KI)I = K(II)$  then  $I = (KI)I = K(II) = KI$  so  $A = IA = KIA = I$  for all  $A$  so the combinatory algebra is trivial.
- (iii) Define  $A_n \equiv (\lambda x_1 \dots x_n. \mathbf{I})$ . Suppose  $A_n = A_{n+m}$  for some  $n, m \geq 1$ , then  $B = A_n \underbrace{II \dots I}_{n+m-1} B = A_{n+m} \underbrace{II \dots I}_{n+m-1} B = I$  for all  $B$ , so the combinatory algebra is trivial unless all the  $A_i$  are distinct. ■

Combinatory algebras also have the following important property (which we have slightly simplified and written in an unusual way, to match our elementary description of combinatory algebras):

**Theorem 5.5.3 (Combinatory Completeness)** Every combinatory algebra is **combinatory complete**. This means that for every sequence of variables  $\langle x_1, \dots, x_n \rangle$ , and every term  $T$  in  $\mathcal{T}_{CL}$  with free variables contained in the sequence, there is an element  $F \in \mathcal{A}$  such that: for each  $A_1, \dots, A_n \in \mathcal{A}$

$$\llbracket T \rrbracket_\rho = F \bullet A_1 \bullet A_2 \cdots \bullet A_n$$

where  $\rho : x_i \mapsto A_i$ .

(This means that all equations in a similar form to  $\exists K. \forall X, Y \in \mathcal{A}. K \bullet X \bullet Y = X$  have solutions for  $K$ ; this generalises the  $K$  and  $S$  axioms.)

**Proof** The element  $F$  is just  $(\lambda x_1 \dots x_n. T)$  (where the brackets indicate the standard translation from a (closed) CL-term to an element of the combinatory algebra). ■

## 5.6 A Simple Combinatory Algebra

Since combinatory algebras have unusual properties, we present a very simple example as a demonstration that they exist at all. A set with a single element is a trivial combinatory algebra (with  $\mathbf{K} = \mathbf{S}$ ) but the next simplest examples are the **term models**.

**Definition** Let  $\mathcal{S}$  be the set  $\Lambda$  quotiented by equality in  $\lambda\beta$ . That is, the elements of  $\mathcal{S}$  are the equivalence (w.r.t. equality in  $\lambda\beta$ ) classes. Because equality is a congruence, this is well defined. Because  $\lambda\beta$  is consistent, it has more than one element (in fact it has countably many). Let  $S$  be the equivalence class of  $\mathbf{s}$  and  $K$  the equivalence class of  $\mathbf{k}$ . Then all the combinatory algebra axioms are satisfied.

The translation of a closed term of the  $\lambda$  calculus is just its own equivalence class. So in fact this is a  $\lambda$ -algebra. Of course, it is hardly surprising that the syntax of the  $\lambda$ -calculus is a good model for the  $\lambda$ -calculus, but it does at least demonstrate that nontrivial combinatory algebras exist.

Interestingly, we can perform the same construction starting from  $\Lambda^0$  instead of  $\Lambda$ , as there was no need for the free variables. But the combinatory algebra generated in this way is not “equivalent” in various model-theoretic ways. This is because, up to equivalence class,  $\Lambda^0$  can be generated by finitely many terms — namely  $\mathbf{k}$  and  $\mathbf{s}$  — whereas  $\Lambda$  cannot because there are infinitely many variables.

## 5.7 $\mathcal{P}\omega$ : A Less Simple Combinatory Algebra

The next simplest models, after the trivial one-element combinatory algebra and the term models, are not so simple. Here we present one, using as elementary presentation as possible. The material of this section is **not on the syllabus** and included only for the interested reader.

Combinatory algebras, unlike more familiar mathematical structures such as groups and fields, are hard to construct. This is because of their connection with the  $\lambda$ -calculus, via the abstraction algorithm: apparently, all functions on the domain of the combinatory algebra must itself be representable in the combinatory algebra and this is impossible by Cantor’s Theorem. Indeed, not until the late 1960s were combinatory algebras (other than term models) described. Their construction is at the heart of early domain theory.

In order to avoid a lengthy discursion into domain theory, we will give a completely elementary construction of a combinatory algebra. The model is due to Scott (see various papers by Scott from the 1970s; it was also discovered independently by Plotkin [Pl72]). For a more natural presentation of this model, and some other models, see [Bar84, Ch. 18], but be prepared for a small amount of topology and category theory.

Scott’s solution to the apparant paradox of including a set’s function space within itself is to consider only *continuous* functions. Just as a continuous function over  $\mathbb{R}$  can be uniquely determined by its value on a set of smaller cardinality (e.g.  $\mathbb{Q}$ ), so the continuous functions over some other sets, with

an appropriate topology, can be encoded in a compact enough way for all continuous functions to live within the set itself. That only continuous functions are needed to make a combinatory algebra (and, more generally, that precisely the continuous functions are representable in the  $\lambda$ -calculus) has far-reaching implications for the study of computation.

The domain of the model (the set over which application is defined) is  $\mathcal{P}\omega$ , i.e. the set of all sets of natural numbers (including zero). The natural ordering on  $\mathcal{P}\omega$  is the subset relation.

Some notational conventions (which differ from the literature but make a lot of sense): we will use greek letters like  $\phi$ ,  $\psi$ , etc for functions on  $\mathcal{P}\omega$ , lowercase Roman letters like  $m$ ,  $n$ , for numbers, and uppercase Roman letters like  $X$ ,  $U$  for sets of numbers (members of  $\mathcal{P}\omega$ ). It will also be convenient to write  $F \subseteq^{\text{fin}} X$  to mean that  $F$  is a *finite* subset of  $X$ . Note that we are only interested in **monotone** functions on  $\mathcal{P}\omega$ .

**Definition** A monotone function  $\phi : \mathcal{P}\omega \rightarrow \mathcal{P}\omega$  is **continuous** if, for every  $X \in \mathcal{P}\omega$ ,

$$\phi(X) = \bigcup_{F \subseteq^{\text{fin}} X} \phi(F).$$

More generally, a monotone function  $\phi : \mathcal{P}\omega^k \rightarrow \mathcal{P}\omega$  is **continuous** if, for every  $X_1, \dots, X_k \in \mathcal{P}\omega$ ,

$$\phi(X_1, \dots, X_k) = \bigcup_{\substack{F_i \subseteq^{\text{fin}} X_i \\ \text{for each } i}} \phi(F_1, \dots, F_k).$$

(For those who are topologically aware, this coincides with the Scott topology on the algebraic cpo  $\mathcal{P}\omega$ . The point of this topology is that every continuous function is uniquely defined by its action on *finite* subsets of  $\mathcal{P}\omega$ , and there are only countably many such.) Every monotone function has  $\phi(X) \supseteq \bigcup_{F \subseteq^{\text{fin}} X} \phi(F)$  so we only need to check that  $m \in \phi(X)$  implies  $m \in \phi(F)$  for some  $F \subseteq^{\text{fin}} X$ , for each  $m$  and  $X$ .

An obvious example of a monotone but non-continuous function is  $\phi : \mathcal{P}\omega \rightarrow \mathcal{P}\omega$  where  $\phi(X) = \emptyset$  when  $X$  is finite and  $\phi(X) = X$  when  $X$  is infinite, but there are other, more subtle, examples. By and large most lemmas about  $\mathcal{P}\omega$  will only work for continuous functions.

In order to encode continuous functions on  $\mathcal{P}\omega$  into  $\mathcal{P}\omega$  we need some simple encodings of pairs and finite sets:

**Definition**

- (i) If  $m, n \in \omega$  then  $\langle m, n \rangle = \frac{1}{2}(m+n)(m+n+1) + m$  (this is a bijection between  $\omega^2$  and  $\omega$ ).
- (ii) If  $\{k_1, \dots, k_m\}$  is a finite subset of  $\omega$ , with  $k_1 < k_2 < \dots < k_m$ , then this set is denoted by  $E_n$ , where  $n = \sum_{i=1}^m 2^{k_i}$ . This is a bijection between the finite subsets of  $\omega$ , and  $\omega$ .

Now we construct the operations which will form the combinatory algebra structure:

**Definition**

- (i) For each  $U \in \mathcal{P}\omega$ ,  $\mathbf{fun}(U) : \mathcal{P}\omega \rightarrow \mathcal{P}\omega$  is given by

$$\mathbf{fun}(U)(X) = \{n \mid \exists E_m \subseteq X \text{ with } \langle m, n \rangle \in U\}.$$

- (ii) If  $\psi : \mathcal{P}\omega^k \rightarrow \mathcal{P}\omega$  is continuous then  $G_\psi \in \mathcal{P}\omega$  is given by

$$G_\psi = \{\langle m_1, \langle m_2, \dots, \langle m_k, n \rangle \dots \rangle \rangle \mid n \in \psi(E_{m_1}, \dots, E_{m_k})\}.$$

$G_\psi$  is called the *graph* of  $\psi$ .

- (iii) The applicative structure on  $\mathcal{P}\omega$  is given by

$$X \bullet Y = \mathbf{fun}(X)(Y).$$

**Lemma 5.7.1**

- (i) The projection functions  $\pi_i^k : \mathcal{P}\omega^k \rightarrow \mathcal{P}\omega$  given by  $\pi_i^k : (X_1, \dots, X_k) \mapsto X_i$  are all continuous.
- (ii)  $\mathbf{fun}(U)$  is always continuous.
- (iii) If  $\phi : \mathcal{P}\omega^k \rightarrow \mathcal{P}\omega$  and  $\chi : \mathcal{P}\omega^k \rightarrow \mathcal{P}\omega$  are continuous then so is  $\psi : (X_1, \dots, X_k) \mapsto \mathbf{fun}(\phi(X_1, \dots, X_k))(\chi(X_1, \dots, X_k))$ .
- (iv) If  $\psi : \mathcal{P}\omega^k \rightarrow \mathcal{P}\omega$  is continuous then

$$G_\psi \bullet X_1 \bullet X_2 \cdots \bullet X_k = \psi(X_1, \dots, X_k).$$

**Proof**

- (i) Trivial.
- (ii) We need to check that  $n \in \mathbf{fun}(U)(X)$  implies  $n \in \mathbf{fun}(U)(F)$  for some  $F \subseteq^{\text{fin}} X$ . If  $n \in \mathbf{fun}(U)(X)$  then there exists some  $E_m \subseteq X$  with  $\langle m, n \rangle \in U$ , but since all  $E_m$  are finite that means that there is some  $F$  with  $E_m \subseteq F \subseteq^{\text{fin}} U$  ( $F = E_m$  will do). Hence  $\mathbf{fun}(U)$  is continuous.

(iii) Let  $X_1, \dots, X_k$  be members of  $\mathcal{P}\omega$ , and suppose that

$$n \in \mathbf{fun}(\phi(X_1, \dots, X_k))(\chi(X_1, \dots, X_k)).$$

We must show that there are finite sets  $F_1, \dots, F_k$  with  $F_i \subseteq X_i$  and

$$n \in \mathbf{fun}(\phi(F_1, \dots, F_k))(\chi(F_1, \dots, F_k)).$$

First,  $n \in \mathbf{fun}(\phi(X_1, \dots, X_k))(\chi(X_1, \dots, X_k))$  if there is some  $E_m \subseteq \chi(X_1, \dots, X_k)$  with  $\langle m, n \rangle \in \phi(X_1, \dots, X_k)$ . Now  $E_m$  is finite and  $\chi$  is continuous, so there are finite subsets  $G_1, \dots, G_k$  with  $E_m \subseteq \chi(G_1, \dots, G_k)$ . Also,  $\phi$  is continuous so there are finite subsets  $H_1, \dots, H_k$  with  $\langle m, n \rangle \in \chi(H_1, \dots, H_k)$ . Finally, take  $F_i = G_i \cup H_i$ .

(iv)

$$\begin{aligned} & G_\psi \bullet X_1 \bullet X_2 \cdots \bullet X_k \\ &= (\mathbf{fun}(G_\psi)(X_1)) \bullet X_2 \cdots \bullet X_k \\ &= (\mathbf{fun}(\{\langle m_1, \langle m_2, \dots, \langle m_k, n \rangle \dots \rangle \rangle \mid n \in \psi(E_{m_1}, \dots, E_{m_k})\})(X_1)) \\ &\quad \bullet X_2 \cdots \bullet X_k \\ &= (\{q \mid \exists E_{p_1} \subseteq X_1 \\ &\quad \text{with } \langle p_1, q \rangle \in \{\langle m_1, \langle m_2, \dots, \langle m_k, n \rangle \dots \rangle \rangle \mid n \in \psi(E_{m_1}, \dots, E_{m_k})\}\}) \\ &\quad \bullet X_2 \cdots \bullet X_k \\ &= (\{\langle m_2, \dots, \langle m_k, n \rangle \dots \rangle \mid \exists E_{p_1} \subseteq X_1 \\ &\quad \text{with } n \in \psi(E_{p_1}, E_{m_2}, \dots, E_{m_k})\}) \\ &\quad \bullet X_2 \cdots \bullet X_k \\ &= \dots \\ &= (\{\langle m_3, \dots, \langle m_k, n \rangle \dots \rangle \mid \exists E_{p_1} \subseteq X_1, E_{p_2} \subseteq X_2 \\ &\quad \text{with } n \in \psi(E_{p_1}, E_{p_2}, E_{m_3}, \dots, E_{m_k})\}) \\ &\quad \bullet X_3 \cdots \bullet X_k \\ &= \dots \\ &= (\{n \mid \exists E_{p_1} \subseteq X_1, E_{p_2} \subseteq X_2, \dots, E_{p_k} \subseteq X_k \\ &\quad \text{with } n \in \psi(E_{p_1}, E_{p_2}, \dots, E_{p_k})\}) \\ &= (\{n \mid \exists F_1 \subseteq^{\text{fin}} X_1, F_2 \subseteq^{\text{fin}} X_2, \dots, F_k \subseteq^{\text{fin}} X_k \\ &\quad \text{with } n \in \psi(F_1, F_2, \dots, F_k)\}) \\ &= \bigcup_{F_i \subseteq^{\text{fin}} X_i} \psi(F_1, \dots, F_k) \\ &= \psi(X_1, \dots, X_k) \text{ because } \psi \text{ is continuous} \end{aligned}$$

■

Finally, we can construct the terms  $K$  and  $S$  to make  $\mathcal{P}\omega$  into a combinatory algebra.

**Definition**

(i) Let  $\kappa : \mathcal{P}\omega^2 \rightarrow \mathcal{P}\omega$  be given by

$$\kappa(X, Y) = X;$$

$\kappa$  is continuous by Lemma 5.7.1(i). Set  $K = G_\kappa$ .

(ii) Let  $\sigma : \mathcal{P}\omega^3 \rightarrow \mathcal{P}\omega$  be given by

$$\sigma(X, Y, Z) = \mathbf{fun}(\mathbf{fun}(X)(Z))(\mathbf{fun}(Y)(Z));$$

$\sigma$  is continuous by repeated application of Lemma 5.7.1(i) and (iii). Set  $S = G_\sigma$ .

**Theorem 5.7.2** With application given by  $\bullet$ , along with  $K$  and  $S$  as above,  $\mathcal{P}\omega$  forms a combinatory algebra.

**Proof** Obvious, given Lemma 5.7.1(iv). ■

It is also true that  $\mathcal{P}\omega$  is a  $\lambda$ -algebra, but the proof requires technology which we have avoided.

So  $\mathcal{P}\omega$  is a “good” model of the untyped  $\lambda$ -calculus, as terms are translated into combinatory logic and hence into elements of  $\mathcal{P}\omega$  (if closed terms, anyway). In fact, the construction  $G_\psi$  gives a way to translate some terms of the  $\lambda$ -calculus directly into  $\mathcal{P}\omega$ , following the same paradigm as used for  $\mathbf{k}$  and  $\mathbf{s}$ . For example, let  $\iota : \mathcal{P}\omega \rightarrow \mathcal{P}\omega$  be the identity map; then  $\llbracket \lambda x.x \rrbracket = G_\iota$ , which turns out (after some tedious computation) to be the set

$$\{\langle m, n \rangle \mid n \in E_m\}.$$

(Can you see why this technique does not work for all terms, and how to fix it?)

It is not obvious that this should necessarily give the same result of translating  $\lambda x.x$  to  $\mathbf{SKK}$ , and so to  $S \bullet K \bullet K$ , and the general proof is messy. But it is true. For the same reasons,  $\llbracket \lambda x.xx \rrbracket = G_\delta$ , where  $\delta : \mathcal{P}\omega^2 \rightarrow \mathcal{P}\omega$  is  $\delta(X) = X \bullet X$ ; some (very boring) calculations tell us that this is the set

$$D = \{\langle m, n \rangle \mid \exists E_{m'} \subseteq E_m \text{ with } \langle m', n \rangle \in E_m\}.$$

We can deduce an interesting fact:

**Lemma 5.7.3** In  $\mathcal{P}\omega$ ,  $\llbracket \Omega \rrbracket = \emptyset$ .

**Proof**  $\llbracket \Omega \rrbracket = D \bullet D$ . Let  $n \in D \bullet D$ . Then  $n \in \mathbf{fun}(D)(D)$ , which

means  $\exists E_p \subseteq D$  with  $\langle p, n \rangle \in D$ . Now looking at the second  $D$ , we have  $\exists E_{p'} \subseteq E_p \subseteq D$  with  $\langle p', n \rangle \in E_p$ .

Choose the least such  $p$ . Now the inclusion means that  $\langle p', n \rangle \in D$ , so  $\exists E_{p''} \subseteq E_{p'}$  with  $\langle p'', n \rangle \in E_{p'}$ . Since  $E_{p''} \subseteq E_{p'}$  we must have  $p'' \leq p'$  (this is routine to check from the definition of the sets  $E_m$ ).

But  $p$  is supposed to be the least number with the property that  $\exists E_{p'} \subseteq E_p \subseteq D$  with  $\langle p', n \rangle \in E_p$ , and we have shown that  $p'$  also has this property (with  $p''$  for  $p'$ ) so we can deduce that  $p = p'$ . Going back to the start, we have that  $n \in D \bullet D$  if  $\langle p, n \rangle \in E_p$ . But  $p \leq \langle p, n \rangle$  (an elementary property of the encodings of pairs) and if  $\langle p, n \rangle \in E_p$  then  $\langle p, n \rangle < p$  (an elementary property of the encodings of sets) so we reach the contradiction  $p < p$ . So  $D \bullet D$  must be empty. ■

In fact, for all unsolvable terms  $s$  we have  $\llbracket s \rrbracket = \emptyset$  in  $\mathcal{P}\omega$ ; there are a number of proofs, including Wadsworth's classic "approximation" method (see [Bar84, §19.1] for one exposition). In many models of the  $\lambda$ -calculus, the unsolvable terms are all denoted by a least element.

## Computational Practice

**5.a** Compute:

- (i)  $\lambda x.yx$ .
- (ii)  $\lambda xy.x$ .
- (iii)  $\lambda xy.yx$ .

**5.b** Which CL terms are the following?

- (i)  $(\lambda xy.yx)_{cl}$ .
- (ii)  $(\lambda x.x(\lambda y.y))_{cl}$ .
- (iii)  $((\lambda x.x)y)_{cl}$ .

**5.c** Which  $\lambda$  term is  $(\mathbf{SS})_\lambda$ ? What is its  $\beta$ -normal form? Which familiar  $\lambda$  term is equal to  $((\mathbf{SII})(\mathbf{SII}))_\lambda$ ?

**5.d** Find the  $\rightarrow_w$  normal forms of:

- (i) **SKIKS**.
- (ii) **SSSS**.
- (iii) **SSS(KKK)SSS**.

## Exercises

**5.1** Compute:

- (i)  $\lambda xy.xy$ .
- (ii)  $\lambda xy.Kxy$ .

**5.2** For any set of terms  $\mathcal{L}$  the set of terms **generated** by  $\mathcal{L}$  is written  $\mathcal{L}^+$  and defined by:

$$\frac{}{s \in \mathcal{L}^+} (s \in \mathcal{L}) \qquad \frac{s \in \mathcal{L}^+ \ t \in \mathcal{L}^+}{st \in \mathcal{L}^+}$$

(Remember that this means that  $\mathcal{L}^+$  is the *least* set satisfying the above rules.)

We say that  $\mathcal{L}$  is a **basis** if for all  $s \in \Lambda^0$ , there is  $t \in \mathcal{L}^+$  with  $\lambda\beta \vdash s = t$ .

Prove that  $\{\mathbf{k}, \mathbf{s}\}$  is a basis.

[Hint: Use Lemma 5.3.1(ii).]

**5.3** Using the above exercise, show that:

- (i)  $\{\theta\}$  is a basis, where  $\theta \equiv \lambda x.x\mathbf{k}\mathbf{s}\mathbf{k}$ . [Hint: Calculate  $\theta\theta\theta$  and  $\theta(\theta\theta)$ .]
- (ii)  $\{X\}$  is a basis, where  $X \equiv \lambda x.x(xs(\mathbf{k}\mathbf{k}))\mathbf{k}$ .

**5.4** Prove all parts of Lemma 5.3.1.

*The next 4 exercises prove that strengthened versions of CL give rise to equivalent equational theories to  $\lambda\beta$  and  $\lambda\beta\eta$ .*

**5.5** Prove

- (i) (a substitution lemma) for  $s, t \in \Lambda$ ,  $(s[t/x])_{cl} \equiv s_{cl}[t_{cl}/x]$ ;
- (ii)  $CL + \text{WExt} + \text{K-Ext} + \text{S-Ext} \vdash \mathbf{K} = \lambda xy.x$ ;
- (iii)  $CL + \text{WExt} + \text{K-Ext} + \text{S-Ext} \vdash \mathbf{S} = \lambda xyz.xz(yz)$ .

**5.6** For all terms  $s, t \in \Lambda$  and  $A, B \in \mathcal{T}_{CL}$ ,

- (i)  $\lambda\beta \vdash (s_{cl})_\lambda = s$ .
- (ii)  $CL + \text{WExt} + \text{K-Ext} + \text{S-Ext} \vdash (A_\lambda)_{cl} = A$ .
- (iii)  $\lambda\beta \vdash s = t$  if and only if  $CL + \text{WExt} + \text{K-Ext} + \text{S-Ext} \vdash s_{cl} = t_{cl}$ .
- (iv)  $CL + \text{WExt} + \text{K-Ext} + \text{S-Ext} \vdash A = B$  if and only if  $\lambda\beta \vdash A_\lambda = B_\lambda$ .



[Hint: (i) is part of Lemma 5.3.1. (ii) is a straightforward induction on  $A$ , using the previous exercise. (iii  $\Rightarrow$ ) is by induction on the proof  $\lambda\beta \vdash s = t$ . (iv  $\Rightarrow$ ) is also by induction on the proof. (iii  $\Leftarrow$ ) follows from (iv  $\Rightarrow$ ) and (i), and (iv  $\Leftarrow$ ) follows from (iii  $\Rightarrow$ ) and (ii).]

**5.7** Show that that extensionality axiom implies the weak extensionality axiom and also the axioms K-Ext and S-Ext.

Can you find an example to show that the converse does not hold?

**5.8** Prove the correspondence result between  $CL + \text{Ext}$  and  $\lambda\beta\eta$ . That is,

- (i)  $\lambda\beta\eta \vdash (s_{cl})_\lambda = s$ .
- (ii)  $CL + \text{Ext} \vdash (A_\lambda)_{cl} = A$ .
- (iii)  $\lambda\beta\eta \vdash s = t$  if and only if  $CL + \text{Ext} \vdash s_{cl} = t_{cl}$ .
- (iv)  $CL + \text{Ext} \vdash A = B$  if and only if  $\lambda\beta\eta \vdash A_\lambda = B_\lambda$ .

(Note that some of these results are immediate consequences of the previous exercises, and the others are simple modifications to the equivalent results of Exercise 5.6.)

**5.9** Prove Lemma 5.5.1. Be careful to cover both implications (separately).

**\*5.10** Here is another example of a combinatory algebra. To construct it, you need to know some computability theory. It is known as **Kleene's first model**, and written  $\mathbf{K}_1$ .

Recall from computability theory that the partial recursive functions can be enumerated. Write  $\{n\}$  for the  $n^{\text{th}}$  partial recursive function.

Let  $\mathbb{N}_\perp = \mathbb{N} \cup \{\perp\}$ , where  $\perp$  is some distinguished symbol. For  $x, y$  in  $\mathbb{N}_\perp$ , define

$$x \cdot y = \begin{cases} n, & \text{if } x, y \in \mathbb{N} \text{ and } \{x\}(y) \text{ is defined and equal to } n, \\ \perp, & \text{otherwise.} \end{cases}$$

Show that  $\mathbb{N}_\perp$  is a combinatory algebra. [Hint: Use the S-m-n Theorem to construct  $K$  and  $S$ .]

**\*5.11** Using Lemma 5.7.3 show that, in  $\mathcal{P}\omega$ ,  $[[\lambda x.\Omega]] = \emptyset$  and  $[[\Omega t]] = \emptyset$ , for any term  $t$ .

This proves that  $\mathcal{P}\omega$  equates strictly more terms than  $\lambda\beta$  (i.e. that some non-equal terms are mapped to equal elements of  $\mathcal{P}\omega$ ).



## Chapter 6

# Simple Types

---

**Reading:** [Hin97] Ch. 2; [Bar92] 3.1

---

We now consider simple type systems for the  $\lambda$ -calculus. Types go back a long way, even as far as the start of the 20th century, where they were used by philosophers to avoid logical paradoxes in the study of mathematical foundations. In the 1930s they were first introduced into combinatory logic and the  $\lambda$ -calculus, and in the 1970s made a big impact in computer programming. In each case, the aim is the same: to prohibit certain “unsafe” operations, more strictures are placed on construction rules (whether for logical sentences, terms, or programs). In the case of logic, the unsafe operations are those which construct logical paradoxes; in the case of programming language they are (usually!) operations where one format of data is used in a way intended for another; in the  $\lambda$ -calculus and simple type systems, the “unsafe” operations are some cases of self-application (indirectly, those which construct terms with infinite reduction paths).

Informally, we will introduce a set of abstract types and a system for showing that terms of the  $\lambda$ -calculus have a certain type. A term  $s$  of type  $A \Rightarrow B$  works like a function from terms of type  $A$  to terms of type  $B$ , and we will only allow an application  $st$  if  $t$  is a term of the correct “input” type  $A$ . Actually, we will not disallow the application (Church’s simple type system did disallow such application altogether) but we will not admit that the application  $st$  has any type if  $t$  is not a term of type  $A$ .

There are very many type systems, and the one we shall examine is arguably the simplest. For an interesting overview of other possible type systems, see [Bar92].

## 6.1 Simple Type Assignment

We will follow the Hindley’s presentation in [Hin97], which is the same in concept but rather different in detail to the classic text [HS86]. It is sometimes known as simple types *a la Curry* and also as  $\text{TA}_\lambda$ , and we will use the latter name.

**Definition** We assume a countable set  $\mathcal{TV}$  of **type variables**, distinct from the object variables of terms. (From now on we will say “term variable” and “type variable” so that the two are not confused.) Type variables will have letters like  $a, b, c, b', a_i$ , etc. From them we build the set of **types**,  $\mathbf{Typ}$ , using the rules:

$$\frac{}{a \in \mathbf{Typ}} \quad (a \in \mathcal{TV}) \qquad \frac{A \in \mathbf{Typ} \quad B \in \mathbf{Typ}}{(A \Rightarrow B) \in \mathbf{Typ}}$$

We will use uppercase Roman letters like  $A, B, C$ , etc, to range over types<sup>1</sup>. The symbol  $\equiv$  will mean equality of types.

Types which are type variables are sometimes called **atomic** types. More practically-oriented type theories usually base their type structure on fundamental computing data types, e.g.  $\mathbb{N}$  the set of natural numbers and/or a two-element boolean type. They are called the **ground types**. Because we are not dealing with concrete computation, we use the type variables as ground types.

Types of the form  $A \Rightarrow B$  are called **composite types**, **function types**, or **arrow types**. We can omit parentheses in arrow types by removing outer parentheses and using the convention that, unless otherwise bracketed, nested arrows associate to the **right**. So  $A \Rightarrow B \Rightarrow C \equiv (A \Rightarrow (B \Rightarrow C))$ . (To understand why application associates to the left and arrow types to the right, note that the “input” side of a type is written on the left of the arrow, but the “input” to a function is written on the right of the application.)

Note: it is useful to observe that every type has finitely many arrows in it. Thus there is no type  $A$  satisfying  $A \equiv A \Rightarrow B$ .

**Definition** A **type context** is a finite (possibly empty) set of pairs  $\Gamma = \{x_1:A_1, \dots, x_n:A_n\}$ , where each  $x_i$  is a term variable and each  $A_i$  a type. The  $x_i$  must be distinct. We use uppercase Greek letters like  $\Gamma, \Delta, \Delta'$ , etc, to range over type contexts. When  $x:A$  is in  $\Gamma$  we say that the term variable  $x$  is “assigned the type  $A$  in  $\Gamma$ ”.

Two contexts  $\Gamma$  and  $\Delta$  are **consistent** if they do not disagree on assignment to any term variable, i.e. if  $\Gamma \cup \Delta$  is a valid context.

<sup>1</sup>Hindley uses lowercase Greek letters like  $\sigma, \tau$ , in [Hin97].

When  $\Gamma = \{x_1:A_1, \dots, x_n:A_n\}$  is a type context we write  $Subjects(\Gamma)$  for the set of term variables  $\{x_1, \dots, x_n\}$ . We also write  $\Gamma - y$  for the type context with any assignment  $y:A$  removed (if  $y \notin Subjects(\Gamma)$  then  $\Gamma - y$  is the same as  $\Gamma$ ). Finally, if  $s$  is a term then we write  $\Gamma \upharpoonright s$  to mean that *all* assignments *except* to free variables of  $s$  are removed from  $\Gamma$ .

The **terms** of  $TA_\lambda$  are the terms of the  $\lambda$ -calculus. We define two ternary relations between contexts, terms, and types, as follows:

**Definition** The relation  $\mapsto$  is written infix as  $\Gamma \mapsto s : A$  and defined by the rules

$$\begin{aligned} \text{(variable)} \quad & \frac{}{\{x:A\} \mapsto x : A} \\ \text{(application)} \quad & \frac{\Gamma \mapsto s : B \Rightarrow A \quad \Delta \mapsto t : B}{\Gamma \cup \Delta \mapsto st : A} \quad (\Gamma \text{ consistent with } \Delta) \\ \text{(abstraction)} \quad & \frac{\Gamma \mapsto s : A}{\Gamma - x \mapsto (\lambda x.s) : B \Rightarrow A} \quad (\{x:B\} \text{ consistent with } \Gamma) \end{aligned}$$

The relation  $\vdash$  is written infix as  $\Gamma \vdash s : A$ , and we define it by

$$\Gamma \vdash s : A \text{ if there is some } \Gamma' \subseteq \Gamma \text{ with } \Gamma' \mapsto s : A.$$

When  $\Gamma$  is the empty context (no term variables assigned to any type) then we can omit it altogether and just write  $\mapsto s : A$  or  $\vdash s : A$ .

The application rule is also known as the  $\Rightarrow$ -elimination rule, and the abstraction rule as  $\Rightarrow$ -introduction. It can be very convenient to split the latter into two cases, depending on whether  $x$  occurs free in  $s$  or not:

$$\begin{aligned} \text{(abstraction-main)} \quad & \frac{\Gamma \cup \{x:B\} \mapsto s : A}{\Gamma \mapsto (\lambda x.s) : B \Rightarrow A} \quad (x \notin Subjects(\Gamma)) \\ \text{(abstraction-vac)} \quad & \frac{\Gamma \mapsto s : A}{\Gamma \mapsto (\lambda x.s) : B \Rightarrow A} \quad (x \notin Subjects(\Gamma)) \end{aligned}$$

Here are some notes about this type system, in comparison to other presentations of simple types: firstly, our type contexts are sets as opposed to sequences. In presentations aimed to mirror more closely the implementation of types on computers, it is common for the information in a type context  $\Gamma$  to come in an order, and there have to be additional rules to allow the information to be moved around inside the context. We view this as an unnecessary complication.

A second, important, choice is that the deduction relation  $\mapsto$  does **not** validate a rule known as **weakening**:

$$\text{(weakening)} \quad \frac{\Gamma \mapsto s : A}{\Gamma' \mapsto s : A} \quad (\Gamma \subseteq \Gamma')$$

This rule, when included, says that not all of the information in  $\Gamma$  need necessarily be used in a deduction. We have omitted this rule, instead making a distinction between the relation  $\mapsto$  (which has been constructed so that only relevant information is allowed in  $\Gamma$  – see below) and the relation  $\vdash$  which does support weakening. This idea comes from Hindley in [Hin97], because it makes some of the following technical proofs a little simpler. Nonetheless it would be more common to have a single relation with weakening included, perhaps implicitly by modification of the other rules.

In this system, only relevant information is allowed in  $\Gamma$ . More precisely,

**Lemma 6.1.1** If  $\Gamma \mapsto s : A$  then  $\text{Subjects}(\Gamma) = \text{FV}(s)$ . If  $\Gamma \vdash s : A$  then  $\Gamma \upharpoonright s \mapsto s : A$ .

**Proof** Obvious, but formally it is by induction on the deduction of  $\Gamma \mapsto s : A$ . ■

Here are some examples of type deductions (fill in for yourself which rule is used at which step):

$$\frac{\frac{}{\{x:A\} \mapsto x : A}}{\mapsto (\lambda x.x) : A \Rightarrow A}}$$

$$\frac{\frac{\frac{}{\{x:A\} \mapsto x : A}}{\{x:A\} \mapsto (\lambda y.x) : B \Rightarrow A}}{\mapsto (\lambda xy.x) : A \Rightarrow B \Rightarrow A}}$$

On the other hand, there is no type  $A$  or context  $\Gamma$  such that  $\Gamma \vdash \lambda x.xx : A$  (we say that  $\lambda x.xx$  is **untypable**; typability is discussed further in Chapter 7). To prove this, suppose that a deduction

$$\Gamma \mapsto \lambda x.xx : A$$

exists. The last step must use the abstraction rule, with premise

$$\{x:B\} \mapsto xx : C$$

for some types  $B$  and  $C$ ; now this can only be deduced using the application rule, with premises

$$\{x:B\} \mapsto x : D \Rightarrow C \text{ and } \{x:B\} \mapsto x : D.$$

Now the variable rule forces  $B \equiv D \Rightarrow C$  and  $B \equiv D$ . But this is impossible, because it requires  $B \equiv B \Rightarrow C$  and no type  $B$  can have this property.

## 6.2 Lemmas About $TA_\lambda$

We prove some basic results about the deduction system  $TA_\lambda$ . The following is particularly useful in proving facts about deductions and we will use it extensively.

**Lemma 6.2.1 (Subject Construction)** Consider the deduction tree of  $\Gamma \mapsto s : A$ .

- (i) The shape of the deduction tree is exactly the same as of the construction tree of  $s$  (upside down).
- (ii) If  $\Delta \mapsto t : B$  appears at some point of this deduction tree, then the subtree rooted at the corresponding node in the construction tree of  $s$  is the construction tree of  $t$ .
- (iii) If  $s$  is a variable, say  $s \equiv x$ , then  $\Gamma$  is necessarily  $\{x:A\}$  for some type  $A$ , and the whole deduction is

$$\frac{}{\{x:A\} \mapsto x : A}$$

- (iv) If  $s$  is an application, say  $s \equiv pq$ , then the final step of the deduction of  $\Gamma \mapsto s : A$  must use the application rule and be of the form

$$\frac{\Gamma \upharpoonright p \mapsto p : B \Rightarrow A \quad \Gamma \upharpoonright q \mapsto q : B}{\Gamma \mapsto pq : A}$$

- (v) If  $s$  is an abstraction, say  $s \equiv \lambda x.p$ , then  $A$  must be of the form  $C \Rightarrow D$  and either

- (a)  $x \in \text{FV}(p)$ , in which case the final step of the deduction of  $\Gamma \mapsto s : A$  is of the form

$$\frac{\Gamma \cup \{x:C\} \mapsto p : D}{\Gamma \mapsto (\lambda x.p) : C \Rightarrow D}$$

or

- (b)  $x \notin \text{FV}(p)$ , in which case the final step of the deduction of  $\Gamma \mapsto s : A$  is of the form

$$\frac{\Gamma \mapsto p : D}{\Gamma \mapsto (\lambda x.p) : C \Rightarrow D}$$

**Proof** Immediate from the definition of  $\Gamma \mapsto s : A$  (notice that the con-

clusions of the rules are exclusive) and Lemma 6.1.1. ■

This result tells us that the structure of  $s$  determines much of the deduction  $\Gamma \mapsto s : A$ . However, not quite all is determined. For example, here is a deduction of  $\mapsto (\lambda xy.y)(\lambda z.z) : A \Rightarrow A$ :

$$\frac{\frac{\frac{\overline{\{y:A\} \mapsto y : A}}{\mapsto (\lambda y.y) : A \Rightarrow A}}{\mapsto (\lambda xy.y) : (B \Rightarrow B) \Rightarrow A \Rightarrow A} \quad \frac{\frac{\overline{\{z:B\} \mapsto z : B}}{\mapsto (\lambda z.z) : B \Rightarrow B}}{\mapsto (\lambda xy.y)(\lambda z.z) : A \Rightarrow A}}$$

Not only is the type  $A$  arbitrary (the same deduction works with any type for  $A$ ) but also the type  $B$ , which does not appear anywhere in the conclusion, is arbitrary.

However, it is the case that deductions *are* unique when the term in the conclusion is a  $\beta$ -normal form. See Exercises 6.7 and 6.8. It is also true, but harder to prove, that deductions are unique for  $\lambda I$ -terms (see Section 2.3 for the definition of a  $\lambda I$ -term).

Another important result is that that  $\alpha$ -equivalent terms can be swapped in deductions, without affecting anything. Indeed, this result is crucial if we want to continue to consider  $\alpha$ -equivalent terms to be identical.

**Lemma 6.2.2 ( $\alpha$ -invariance)** If  $\Gamma \mapsto s : A$  and  $t \equiv s$  then  $\Gamma \mapsto t : A$ .

**Proof** Omitted because it is fiddly to formalise properly. But it is easy to see how a deduction for  $s$  can be changed into a deduction for  $t$ , just by renaming the bound variables in each node of the deduction tree. ■

Finally, we prove a useful lemma about substitutions in typed terms.

**Lemma 6.2.3** If  $\Gamma \cup \{x:B\} \mapsto s : A$  and  $\Delta \mapsto t : B$ , with  $\Gamma$  consistent with  $\Delta$ , and  $x$  is not free in  $t$ , then  $\Gamma \cup \Delta \mapsto s[t/x] : A$ .

**Proof** By induction on  $s$ .

If  $s$  is a variable then it must be the variable  $x$  (why?),  $\Gamma$  is empty, and  $A \equiv B$ . The deduction  $\Delta \mapsto t : B$  is the deduction  $\Gamma \cup \Delta \mapsto s[t/x] : A$ .

If  $s \equiv pq$ , by the Subject Construction Lemma 6.2.1 the final step in the deduction  $\Gamma \mapsto s : A$  must be

$$\frac{(\Gamma \cup \{x:B\}) \upharpoonright p \mapsto p : C \Rightarrow A \quad (\Gamma \cup \{x:B\}) \upharpoonright q \mapsto q : C}{\Gamma \cup \{x:B\} \mapsto pq : A}$$

Now either



- (i)  $x$  occurs free in  $p$  but not  $q$ . Then  $(\Gamma \cup \{x:B\}) \upharpoonright p = \Gamma \upharpoonright p \cup \{x:B\}$  and  $(\Gamma \cup \{x:B\}) \upharpoonright q = \Gamma \upharpoonright q$ . Applying the inductive hypothesis to the former, we have  $\Gamma \upharpoonright p \cup \Delta \mapsto p[t/x] : A$ . Now using the application rule, we deduce

$$\Gamma \upharpoonright p \cup \Gamma \upharpoonright q \cup \Delta \mapsto p[t/x]q : A$$

(note that the contexts are indeed compatible, because  $\Gamma$  is compatible with  $\Delta$ ); the above context is exactly  $\Gamma \cup \Delta$  and the above term is  $(pq)[t/x]$ , which is the result we need.

- (ii)  $x$  occurs free in  $q$  but not  $p$ . Symmetrical to above.  
 (iii)  $x$  occurs free in both  $p$  and  $q$ . This time we have  $(\Gamma \cup \{x:B\}) \upharpoonright p = \Gamma \upharpoonright p \cup \{x:B\}$  and  $(\Gamma \cup \{x:B\}) \upharpoonright q = \Gamma \upharpoonright q \cup \{x:B\}$  so we apply the inductive hypothesis to both  $p$  and  $q$ . The rest of the case is as (i).

(Note that  $x$  must occur free in at least one of  $p$  or  $q$ .)

The final case,  $s \equiv \lambda y.p$ , is done in Exercise 6.5. We must have that  $y$  is fresh (not equal to  $x$  or free in  $t$ ), but  $\alpha$ -conversion can ensure this. ■

### 6.3 The Subject Reduction Theorem

Any type system should behave properly with respect to reduction. If terms which receive no type are considered “unsafe”, then it would be worrying for “safe” terms to reduce to “unsafe” terms (although the converse might happen). Many type systems have the property known as **subject reduction**, including  $TA_\lambda$ :

**Theorem 6.3.1 (Subject Reduction)** If  $\Gamma \vdash s : A$  and  $s \rightarrow_\beta t$  then  $\Gamma \vdash t : A$ .

(Note that this is in terms of  $\vdash$  and not the stronger relation  $\mapsto$ . See Exercise 6.4.)

**Proof** First, note that we have  $\Gamma' \mapsto s : A$ , where  $\Gamma' = \Gamma \upharpoonright s$ . Use induction on the proof of  $s \rightarrow_\beta t$ .

Suppose  $s \rightarrow_\beta t$  is by reduction on the right of an application,

$$\frac{p \rightarrow_\beta q}{s \equiv up \rightarrow_\beta uq \equiv t}$$

By the Subject Construction Lemma 6.2.1 we know that the final step in the deduction  $\Gamma' \mapsto s : A$  must be

$$\frac{\Gamma' \upharpoonright u \mapsto u : B \Rightarrow A \quad \Gamma' \upharpoonright p \mapsto p : B}{\Gamma' \mapsto up : A}$$

The inductive hypothesis applies to  $\Gamma' \vdash p : B$ , telling us that  $\Gamma' \vdash q : B$ . So we have

$$\frac{\Gamma' \upharpoonright u \mapsto u : B \Rightarrow A \quad \Gamma' \upharpoonright q \mapsto q : B}{\Gamma' \mapsto uq : A}$$

as required.

The case when  $s \rightarrow_\beta t$  is by reduction on the left of an application is completely symmetrical.

The abstraction case is also similar, in that the inductive hypothesis allows the relevant deduction to be constructed. This is left as an exercise.

Finally, there is the case when  $s \equiv (\lambda x.p)q$  and  $t \equiv p[q/x]$ . By the Subject Construction Lemma 6.2.1, used twice, we know that the final steps in the deduction  $\Gamma' \mapsto s : A$  must be

$$\frac{\frac{\Gamma' \upharpoonright (\lambda x.p) \cup \{x:B\} \mapsto p : A}{\Gamma' \upharpoonright (\lambda x.p) \mapsto \lambda x.p : B \Rightarrow A} \quad \Gamma' \upharpoonright q \mapsto q : B}{\Gamma' \mapsto (\lambda x.p)q : A}$$

Now Lemma 6.2.3 applies to  $\Gamma' \upharpoonright (\lambda x.p) \cup \{x:B\} \mapsto p : A$  and  $\Gamma' \upharpoonright q \mapsto q : B$ , telling us that  $\Gamma' \upharpoonright ((\lambda x.p)q) \mapsto p[q/x] : A$ ; this is the term  $t$  and the context restriction can be lifted by weakening. ■

This shows that, when  $s \rightarrow_\beta t$  (or indeed  $s \twoheadrightarrow_\beta t$ ), then any type which  $s$  has,  $t$  also has. The converse result is known as **subject expansion**: if  $\Gamma \vdash t : A$  and  $s \rightarrow_\beta t$  then  $\Gamma \vdash s : A$ ; subject expansion does not hold in this type system. See for example Exercise 6.6; also note that no term containing  $\Omega$  as a subterm can have any type, but  $(\lambda xy.y)\Omega$  reduces to  $\lambda y.y$ , which does. There is even an example (see [Hin97, 7A2.1]) where the reduct has no types at all in common with the original term, although both do have some types.

It is a fact that there are partial converses to the subject reduction result. For example: if  $s$  reduces to  $t$  by non-cancelling and non-duplicating reductions (see Section 2.3) and  $\Gamma \vdash t : A$  then  $\Gamma \vdash s : A$ . But there is no general converse.

As a consequence, the property that a term has a specific type is not closed under  $(\lambda\beta)$  equality. This is unsatisfactory from a theoretical point of view, if equal terms are supposed to represent equal functions (although it has not proved to be any problem from a practical standpoint). A natural extension to  $\text{TA}_\lambda$  is to add an equality rule, in addition to the others for type inference:

$$\text{(equality)} \quad \frac{\Gamma \mapsto s : A \quad \lambda\beta \vdash s = t}{\Gamma \mapsto t : A}$$

In this system, which Hindley calls  $\text{TA}_{\lambda+\beta}$ , some terms which previously had no type now do have a type, for example  $(\lambda xy.y)\Omega$ . This system is discussed

thoroughly in Chapter 4 of [Hin97] and it is explored a little further in the exercises. There are some initial difficulties (not least that the Subject Construction Lemma breaks down) which can largely be overcome. However, in contrast to  $TA_\lambda$ , the relation  $\Gamma \vdash s : A$  is undecidable in  $TA_{\lambda+\beta}$  (we shall see that it is decidable in  $TA_\lambda$  in Chapter 8). See Exercise 7.4.

## 6.4 Equality and Reduction in $TA_\lambda$

The type system  $TA_\lambda$  stands independently of the properties of the terms themselves. The standard features of the untyped language are unchanged in the typed world:

- We can axiomatise equality with the same rules as  $\lambda\beta$ . Equality is still consistent, for the same reasons as before.
- Reduction has the same definition, and the Subject Reduction Theorem ensures that reduction interacts properly with the type system.
- The Church-Rosser proof carries over. Because of Subject Reduction, we know that the term  $u$  at the bottom of the diamond can be given the same type as the term  $s$  at the top.
- Reduction strategies still make sense. Leftmost reduction is still normalizing. However, see the next chapter.

In the next chapter we will restrict our attention to terms which do have some type, and the properties of the  $\lambda$ -calculus thus restricted will be seen to be quite different to the full untyped language.

## 6.5 An Alternative Presentation of Simple Types

Classically there were two different approaches to the simply typed  $\lambda$ -calculus: Curry's version which is similar to that which we have given in this chapter, and Church's "typed terms" presentation. For interest we now give a brief account of the latter. This material is **not on the syllabus**.

Instead of using the auxiliary notion of type deduction to deduce types for (a subset of) the untyped terms, in Church's system the set of terms is modified to include type information. Indeed, there are now many sets of terms, one set for each type.

We suppose that for each type  $A$  there is a countable set of variables of type  $A$ ,  $\mathcal{V}^A$ , including  $x^A, y^A, \dots$

**Definition** For each type  $A$  we define the set of **terms of type  $A$** , written  $\Lambda^A$ , by the following mutually inductive rules:

$$\frac{}{x^A \in \Lambda^A} (x^A \in \mathcal{V}^A) \quad \frac{s \in \Lambda^{A \Rightarrow B} \quad t \in \Lambda^A}{st \in \Lambda^B} \quad \frac{s \in \Lambda^B}{\lambda x^A. s \in \Lambda^{A \Rightarrow B}} (x^A \in \mathcal{V}^A)$$

Most things carry over from the untyped setting: free and bound variables, closed terms, substitution, contexts,  $\alpha$ -conversion (which must preserve types). Terms which are  $\alpha$ -convertible are again considered syntactically equivalent. Contexts have to be limited to accept terms with specified types as parameters.

Note that, quite unlike in Curry's system, all terms have a unique type. The type "tags" on each variable are sufficient immediately to identify the type of a term.

We must redefine reduction to become a family of relations  $\rightarrow_{\beta}^A$ , one for each type  $A$ , denoting  $\beta$ -reduction on terms of each type. This is well-defined because

**Lemma 6.5.1** If  $s \in \Lambda^A$  and  $t \in \Lambda^B$  then  $s[t/x^B] \in \Lambda^A$ .

Of course the Subject Reduction Theorem is now immediate, because reduction  $s \rightarrow_{\beta}^A t$  is only *defined* when  $s$  and  $t$  have the same type. It also means that Subject Expansion now holds. This typed reduction does satisfy the Church-Rosser property, but in fact this cannot be deduced immediately from the result for untyped reduction (the untyped proof can be adapted fairly easily, though).

Many results parallel to those in the type system  $\text{TA}_{\lambda}$  can be proved, including those of the next chapter. But there is a fundamental philosophical difference between the two presentations, which correspond to differences between type systems in real-world programming languages. Church's presentation is akin to typed programming languages where the programmer must specify the type of every object they define; the compiler checks that all uses of typed objects are consistent and fails if they are not. Curry's presentation is akin to typed languages where the programmer need not specify the type of every object; the compiler tries to deduce consistent types for everything and fails if it cannot.

## Computational Practice

**6.a** Write these types with all implicit brackets shown explicitly:

- (i)  $A \Rightarrow A \Rightarrow A \Rightarrow A$ .
- (ii)  $A \Rightarrow (B \Rightarrow C) \Rightarrow A$ .
- (iii)  $A \Rightarrow (B \Rightarrow C) \Rightarrow (A \Rightarrow D) \Rightarrow E$ .

**6.b** Give deductions of the following:

- (i)  $\{y:A\} \mapsto \lambda x.xy : (A \Rightarrow B) \Rightarrow B$ .
- (ii)  $\{x:A \Rightarrow A\} \mapsto \lambda y.xy : A \Rightarrow A$ .
- (iii)  $\{x:B \Rightarrow C\} \mapsto \lambda y.xy : B \Rightarrow C$ .
- (iv)  $\mapsto \lambda zyx.x(yz) : A \Rightarrow (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow C$ .

**6.c** Find a type  $A$  such that there is some deduction of

- (i)  $\vdash (\lambda xyz.y) : A$ .
- (ii)  $\vdash (\lambda xy.yx) : A$ .
- (iii)  $\vdash (\lambda xyz.xyz) : A$ .
- (iv)  $\vdash (\lambda xy.yx)(\lambda x.x) : A$ .

## Exercises

**6.1** Give deductions of the following:

- (i)  $\{x:A\} \vdash (\lambda y.y)x : A$ .
- (ii)  $\vdash \lambda xy.xyy : (A \Rightarrow A \Rightarrow B) \Rightarrow A \Rightarrow B$ .
- (iii)  $\vdash \lambda xy.xy : (A \Rightarrow B) \Rightarrow A \Rightarrow B$ .

**6.2** Show that there is no type  $A$  such that  $\vdash \lambda xy.yxy : A$ .

Is the same true for  $\vdash \lambda xy.yxx : A$ ?

**6.3** Write down closed terms  $s$  which can be given the following types (in other words, find a closed term  $s$  such that  $\mapsto s : A$ ):

- (i)  $(A \Rightarrow A) \Rightarrow A \Rightarrow A$ ,
- (ii)  $(A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C$ ,

Can you find a closed term  $s$  such that  $\mapsto s : a \Rightarrow b$ , for different type variables  $a$  and  $b$ ?

**6.4** Show that  $\{x:A\} \mapsto (\lambda yz.z)x : B \Rightarrow B$  but not  $\{x:A\} \mapsto (\lambda z.z) : B \Rightarrow B$ , so that Theorem 6.3.1 cannot be strengthened to  $\mapsto$  for  $\vdash$ .

Can you find sufficient conditions under which the stronger result does hold?

**6.5** Complete the abstraction case in the proof of Lemma 6.2.3, and also in the proof of Theorem 6.3.1. There will be two subcases, corresponding to abstractions where the abstracted variable does or does not occur in the abstraction body.

**6.6**

- (i) Let  $s \equiv (\lambda xy.y)(\lambda z.zz)$  and  $t \equiv \lambda y.y$ . Show that  $s \rightarrow_\beta t$  and  $\vdash t : A \Rightarrow A$ , but that there is no type  $B$  such that  $\vdash s : B$ .
- (ii) (Harder!) Let  $s \equiv \lambda xyz.(\lambda w.y)(xz)$  and  $t \equiv \lambda xyz.y$ . Show that  $s \rightarrow_\beta t$ ,  $\vdash s : (C \Rightarrow D) \Rightarrow B \Rightarrow C \Rightarrow B$  for any types  $B, C$ , and  $D$ , that  $\vdash t : A$  for some types  $A$ , but that not  $\vdash t : (C \Rightarrow D) \Rightarrow B \Rightarrow C \Rightarrow B$ .

(These are examples where subject expansion fails.)

**6.7** Suppose that  $s$  is a  $\beta$ -normal form and that  $\Gamma \vdash s : A$ . Show that every type which appears as a conclusion anywhere in the deduction tree of  $\Gamma \vdash s : A$  occurs either as a subtype of  $A$  or in  $\Gamma$ .

[Hint: prove the result by induction on  $s$ ]

**6.8** Suppose that  $s$  is a  $\beta$ -normal form and that  $\Gamma \vdash s : A$ . Using the previous exercise and the Subject Construction Lemma show that the deduction is unique.

*The next 3 exercises show that some of the computability power of the untyped  $\lambda$ -calculus is available in the simply typed setting.*

**6.9** Prove by induction that there is a type **int** such that all Church numerals  $\ulcorner n \urcorner$  satisfy  $\vdash \ulcorner n \urcorner : \mathbf{int}$ .

[Hint: Find **int** by looking at Church numerals greater than 1.  $\ulcorner 0 \urcorner$  and  $\ulcorner 1 \urcorner$  are anomalous in that they can be proved to have a wider range of types.]

Also give a deduction that  $\vdash \mathbf{succ} : \mathbf{int} \Rightarrow \mathbf{int}$ .

**6.10** Recall that the projection functions are  $\lambda$  definable by the terms  $\mathbf{p}_m^i \equiv \lambda x_1 \dots x_m.x_i$ . Show that they satisfy  $\vdash \mathbf{p}_m^i : \underbrace{\mathbf{int} \Rightarrow \dots \Rightarrow \mathbf{int}}_{m \text{ times}} \Rightarrow \mathbf{int}$ .

**6.11** Show that  $\mathbf{plus} \equiv \lambda mnfx.mf(nfx)$  and  $\mathbf{times} \equiv \lambda mnfx.m(nf)x$  define the addition and multiplication functions on nonnegative integers (you will need to use induction).

Give deductions showing  $\vdash \mathbf{plus} : \mathbf{int} \Rightarrow \mathbf{int}$  and  $\vdash \mathbf{times} : \mathbf{int} \Rightarrow \mathbf{int}$ .

## Chapter 7

# Typability and Strong Normalization

---

**Reading:** [GLT89] Ch. 6; [Han94] 7.1

---

The simple type system of the previous chapter deduces types for terms. The terms considered are all of the untyped terms, although some of them have no type at all (and, we shall see in the next chapter, if a term has any type at all it has infinitely many). Nonetheless, we retain the language to discuss terms which have no type (this is an advantage of Curry's system over Church's). In this chapter we consider the set of all terms, but examine properties of the subset which do have some type.

### 7.1 Typability

**Definition** Let  $s$  be a term of the untyped  $\lambda$ -calculus. We say that  $s$  is **typable** if there is some context  $\Gamma$  and type  $A$  such that  $\Gamma \vdash s : A$ .

Some examples of typable terms:  $x$ ,  $\mathbf{i}$ ,  $\mathbf{k}$ ,  $\mathbf{s}$ ,  $(\lambda x.x)(\lambda x.x)$ ,  $\ulcorner n \urcorner$ , etc. It is quite easy to prove that a term is typable, by exhibiting a deduction  $\Gamma \vdash s : A$ .

Some examples of untypable terms:  $xx$ ,  $\lambda x.xx$ ,  $\mathbf{\Omega}$ ,  $\mathbf{y}$ , etc. We shall find techniques which can be applied to prove terms untypable subsequently.

If the type system is used to restrict to “safe” terms, then the typable terms are those which are “safe”. We have the following properties of the typable terms:

**Lemma 7.1.1** The set of typable terms is closed under:

- (i) taking subterms,

- (ii) abstraction,
- (iii)  $\beta$ -reduction.

**Proof**

- (i) Lemma 6.2.1(ii).
- (ii) The (abstraction) rule of  $\text{TA}_\lambda$ .
- (iii) Theorem 6.3.1.

■

## 7.2 Tait’s “Reducibility” Proof

All typable terms are strongly normalizing wrt  $\beta$ -reduction: they have no infinite reduction sequence; put another way, they have a  $\beta$ -normal form and it is impossible to avoid finding it, so long as one keeps reducing where possible. This is a hard theorem to prove.

Historically, there was first a proof that all typable terms are weakly normalizing (i.e. they all have  $\beta$ -normal forms) which was sketched by Turing. It involved making a complexity measure on redexes and using a reduction strategy which always reduces the most complex redex; it can be shown that this reduces the overall complexity of the term, which is finite, and hence that this reduction strategy must find a normal form. But the strong normalization theorem implies the weak normalization theorem, so we will not include a proof of the latter.

For the former, we follow Tait’s “reducibility” proof, originally in [Tai67]; we use a version of the proof similar to Hankin’s [Han94, 7.1]. Our strategy is as follows:

- (i) Make an abstract definition of a property called **reducibility**, on typable terms.
- (ii) Prove that every reducible term is strongly normalizing.
- (iii) Prove that every typable term is reducible.

For both (ii) and (iii) we have to prove something a bit stronger, to make the inductions work. (ii) comes from Theorem 7.2.1, which is proved by induction on the type. (iii) comes from Theorem 7.2.3, which is proved by induction on the term.

This technique is a bit mysterious, because the abstract property called reducibility does not seem to have much significance on its own, other than to make this proof work. In fact the same technique can be adapted for so many other versions of  $\lambda$ -calculus, including higher-order type theories, that it is now considered the “standard” approach to proving strong normalization theorems.



The notion of reducibility is defined inductively over *types*:

**Definition** Suppose that  $\Gamma \vdash s : A$ . We say that  $s$  is **reducible** (also called **strongly computable** and nothing to do with either reduction or computability) if it meets these conditions:

- (i) If  $A$  is atomic (a type variable), then  $s$  must be strongly normalizing.
- (ii) If  $A \equiv B \Rightarrow C$ , then for all reducible  $t$  such that  $\Gamma' \vdash st : C$ ,  $st$  must be reducible.

Technical note: reducibility is a property of both the term  $s$  and the type  $A$ . According to the definition, it is possible that a term  $s$  which can be given two types  $A$  and  $B$  might be reducible at  $A$  and not at  $B$ . In fact this does not happen because all terms are reducible but we emphasise that the definition involves both  $s$  and  $A$ .

Note that every type  $A$  can be uniquely decomposed into the form

$$A \equiv A_1 \Rightarrow A_2 \cdots \Rightarrow A_n \Rightarrow a,$$

where  $a$  is a type variable (and possibly  $n = 0$ ). Unpicking the inductive definition a bit, we have that  $s$  is reducible if, for all reducible terms  $t_1, \dots, t_n$  of types  $A_1, \dots, A_n$  respectively,  $st_1 \dots t_n$  is strongly normalizing.

**Theorem 7.2.1** Let  $A$  be a type. Then

- (I) If  $\Gamma \vdash s : A$  and  $s$  is reducible then  $s$  is strongly normalizing.
- (II) If  $\Gamma \vdash vs_1s_2 \dots s_n : A$  (for some  $n \geq 0$ ), with all  $s_i$  strongly normalizing, and  $v$  is any term variable, then  $vs_1s_2 \dots s_n$  is reducible.

**Proof** By induction on the type  $A$ .

- $A$  atomic

- (I) Immediate from the definition of reducible.
- (II) All  $s_i$  are strongly normalizing, and therefore so is  $vs_1 \dots s_n$  because there are no additional redexes created in the application with only a variable at the head. By the definition,  $vs_1 \dots s_n$  is therefore reducible.

- $A \equiv B \Rightarrow C$

- (I) Suppose that  $\Gamma \vdash s : B \Rightarrow C$  with  $s$  reducible. Take a fresh variable  $x$ ; we have a type deduction  $\{x:B\} \vdash x : B$  and, by freshness of  $x$ ,  $\{x:B\}$  is compatible with  $\Gamma$ . Also, the type of  $x$  is shorter than  $A$ ,

so applying (II) of the inductive hypothesis (with  $n = 0$ ) to  $x$  tells us that  $x$  itself is reducible.

Since  $s$  is reducible,  $sx$  must be reducible. But the type of  $sx$  is  $C$ , also shorter than  $A$ , so the inductive hypothesis applies here too. (I) of this inductive hypothesis tells us that  $sx$  is strongly normalizing. Therefore so is  $s$ , because subterms of strongly normalizing terms must themselves be strongly normalizing.

- (II) Suppose  $\Gamma \vdash vs_1 \dots s_n : B \Rightarrow C$  with all  $s_i$  strongly normalizing. Write  $s \equiv vs_1 \dots s_n$ . Let  $t$  be a term, with  $\Gamma' \vdash st : C$  for some  $\Gamma'$ , and suppose that  $t$  is reducible. In order to show  $s$  reducible we need to show  $st$  reducible for all such  $t$ .

Note that  $t$  is of type  $B$ , shorter than  $A$ , so (I) of the inductive hypothesis applies to  $t$ : it tells us that  $t$  is strongly normalizing.

Also  $C$  is a shorter type than  $A$ , so the inductive hypothesis applies to  $st$ ; but  $st$  is also of the form  $vs_1 \dots s_n s_{n+1}$  with all the  $s_i$  strongly normalizing, so (II) of the inductive hypothesis tells us that this term is reducible, as required. ■

We also need a lemma about reducibility of applications and abstractions:

**Lemma 7.2.2**

- (i) If  $\Gamma \vdash s : A \Rightarrow B$  and  $\Gamma' \vdash st : B$ , and  $s$  and  $t$  are both reducible, then  $st$  is reducible.
- (ii) If  $\Gamma \cup \{y:B\} \vdash s : A$  and, for all  $\Gamma'$  consistent with  $\Gamma$  where  $\Gamma' \vdash t : B$  and  $t$  reducible we have  $s[t/y]$  reducible, then  $\lambda y.s$  is reducible.

**Proof**

- (i) Suppose that  $B \equiv B_1 \Rightarrow B_2 \dots \Rightarrow B_n \Rightarrow b$ . To show that  $st$  is reducible we must show that, for all reducible  $u_1, \dots, u_n$  with  $\Delta \vdash stu_1 \dots u_n : b$ ,  $stu_1 \dots u_n$  is strongly normalizing. But using the same “unpicked” version of the definition of reducibility of  $s$ , and the fact that  $t$  is reducible, gives precisely this.
- (ii) Note that  $\Gamma \vdash \lambda y.s : B \Rightarrow A$ . Suppose that  $B \Rightarrow A \equiv A_1 \Rightarrow A_2 \dots \Rightarrow A_n \Rightarrow a$ , and let  $u_1, \dots, u_n$  be reducible terms with  $\Delta \vdash (\lambda y.s)u_1 \dots u_n : a$ . Write  $s' = (\lambda y.s)u_1 \dots u_n : a$ . We must show that  $s'$  is strongly normalizing. Take any reduction sequence from  $s'$ : if it does not reduce the redex  $(\lambda y.s)u_1$  then it must be finite, because it can only involve reductions inside  $s$ , and  $u_1, u_2$ , etc, separately – these terms are all reducible and hence strongly normalizing by Theorem 7.2.1. If the reduction sequence does reduce  $(\lambda y.s)u_1$  (perhaps

after  $s$  and  $u_1$  have been themselves reduced) then we have some reduct of  $s[u_1/y]u_2 \dots u_n$ . But a)  $s[u_1/y]$  is reducible by hypothesis, and b) therefore so is  $s[u_1/y]u_2 \dots u_n$ , because all the  $u_i$  were assumed reducible. Therefore  $s[u_1/y]u_2 \dots u_n$  must be strongly normalizing thanks to Theorem 7.2.1, hence the same for all its reducts. ■

**Theorem 7.2.3** Suppose that  $\{x_1:A_1, \dots, x_n:A_n\} \vdash s : A$  ( $s$  is not assumed reducible). Then for every  $u_1 \dots u_n$  such that  $\Gamma_i \vdash u_i : A_i$  (with  $\cup \Gamma_i$  consistent), if all the  $u_i$  are reducible then so is  $s[u_1/x_1, u_2/x_2, \dots, u_n/x_n]$ .

**Proof** By induction on the structure of  $s$ . We will write  $s[\vec{u}/\vec{x}]$  as shorthand for the simultaneous substitution  $s[u_1/x_1, u_2/x_2, \dots, u_n/x_n]$ .

- $s \equiv x_i$  for some  $i$

(Note that  $s$  cannot be any other variable, by Lemma 6.1.1). Then  $s[\vec{u}/\vec{x}] \equiv u_i$ , which is reducible by hypothesis.

- $s \equiv pq$

By the Subject Construction Lemma 6.2.1, and weakening, we must have  $\Gamma \vdash p : B \Rightarrow A$  and  $\Gamma \vdash q : B$ , for some type  $B$ . The induction hypothesis applies to  $p$  and  $q$ , telling us that  $p[\vec{u}/\vec{x}]$  and  $q[\vec{u}/\vec{x}]$  are both reducible. By Lemma 7.2.2(i), so is  $p[\vec{u}/\vec{x}]q[\vec{u}/\vec{x}]$ , and this is just  $s[\vec{u}/\vec{x}]$ .

- $s \equiv \lambda y.p$

We may assume that  $y$  is not any of the  $x_i$ . By the Subject Construction Lemma 6.2.1 we must have  $\Gamma \cup \{y:B\} \vdash p : C$  with  $A \equiv B \Rightarrow C$ . The induction hypothesis applies to  $p$ , telling us that for all reducible terms  $t$  and  $u_1, \dots, u_n$  we have  $p[\vec{u}/\vec{x}][t/y]$  reducible. Applying Lemma 7.2.2(ii), we have that  $s[\vec{u}/\vec{x}]$  is therefore reducible. ■

**Corollary 7.2.4** Let  $s$  be typable. Then  $s$  is strongly normalizing.

**Proof** Suppose that  $\{x_1:A_1, \dots, x_n:A_n\} \vdash s : A$ . We know that  $\{x_i:A_i\} \vdash x_i : A_i$ , and all  $x_i$  are reducible (by Theorem 7.2.1(II) with  $n = 0$ ). Therefore  $s \equiv s[x_1/x_1, \dots, x_n/x_n]$  is reducible by Theorem 7.2.3. Hence  $s$  is strongly normalizing by Theorem 7.2.1(I). ■

### 7.3 Consequences of Strong Normalization

Strong Normalization marks the simply typed  $\lambda$ -calculus as very different to the untyped version. Firstly,

**Corollary 7.3.1** Equality of typable terms is decidable.

**Proof** To decide whether  $\lambda\beta \vdash s = t$ , when  $s$  and  $t$  are typable, reduce both to normal form and compare.

(Note: one must take care to spot normal terms which are equal by  $\alpha$ -conversion. The simplest way is to rename bound variables canonically, e.g. the first bound variable, reading from the left, is renamed to  $x_1$ , the second to  $x_2$ , and so on. Alternatively, use a “nameless” representation of terms, with bound variables replaced by a “pointer” to the respective binding  $\lambda^1$ .) ■

Also, the strong normalization result gives another way to prove that a term is untypable: simply exhibit an infinite reduction path (but note Exercise 7.1).

More seriously, if we want to use the  $\lambda$ -calculus as a prototypical programming language, we have:

**Corollary 7.3.2** There are no fixed point combinators in  $TA_\lambda$ .

**Proof** By Exercise 7.3 no fixed point combinators have a normal form, so they cannot be typable. ■

So our standard techniques for solving recursive equations in the untyped  $\lambda$ -calculus no longer apply. This has consequences for expressive power, and in particular the definability of functions in the simply typed language.

We can encode numerals into the simply typed  $\lambda$ -calculus as in the untyped language: let **int** be the type from Exercise 6.9, which shows that the standard church numerals are all typable with types given by **int**. Also, the successor combinator **succ** is typable and compatible with application to numerals.

It is possible to prove the following:

**Lemma 7.3.3** The projection functions, constant functions, addition and multiplication are all definable in simply typed  $\lambda$  calculus (w.r.t. Church numerals).

---

<sup>1</sup>An interesting and compelling representation, suggested by de Bruijn: see [dB72] or [Bar84, App. C] for early expositions.

See Exercises 6.9-6.11.

The fact of strong normalization in the simply typed  $\lambda$ -calculus should make us suspect that the language is not Turing Complete — our obvious candidate for undefined is not available. The consequential fact that  $\beta$ -convertibility is decidable is also a pointer. Most importantly we have the lack of fixed point combinators.

It is possible to show that Lemma 7.3.3 is in fact as far as one can go. The following theorem is independently due to Schwichtenberg [Sch76] and Statman [Sta79].

**Theorem 7.3.4** Define the set of functions on the natural numbers called the **extended polynomials** to be the smallest set containing projection functions, constant functions, the function  $sg$  where  $sg : 0 \mapsto 0$  and  $sg : n + 1 \mapsto 1$ , and closed under addition and multiplication of functions.

Then the functions definable in the simply typed  $\lambda$ -calculus are exactly the extended polynomials.

In fact we can get some more definable functions if we do not insist on a single type **int** for numerals, but we still cannot get all total recursive functions, let alone partial recursive functions.

## Computational Practice

There is no computation to practice, in this chapter!

## Exercises

**7.1** Theorem 7.2.4 shows that every term which is typable is strongly normalizing. Show the converse is false, i.e. that there is a term which is strongly normalizing but not typable.

**7.2** Recall the definitions in 2.3, of the subsets of untyped terms called  **$\lambda I$ -terms**, **affine terms**, and **linear terms**.

Without reference to types, give a direct proof that all affine terms are strongly normalizing, and deduce that all linear terms are strongly normalizing (hint: consider the size of the term under  $\beta$ -reduction). By means of a counterexample, show that not all  $\lambda I$ -terms are necessarily strongly normalizing.

**7.3** Show that, if there exists a (untyped) fixed point combinator which has a normal form, then there exists a fixed point combinator which is in normal form.

Deduce that no fixed point combinator has a normal form.

[Hint: consider a fixed point combinator  $F$  in normal form and suppose that  $Fx = x(Fx)$ , for a fresh variable  $x$ . By considering reducts of  $Fx$  and  $x(Fx)$ , derive a contradiction.]

*The next 3 exercises concern the type system with equality  $TA_{\lambda+\beta}$ , mentioned briefly in the previous chapter. It has all the rules of  $TA_{\lambda}$ , plus the additional type deduction rule*

$$\text{(equality)} \quad \frac{\Gamma \mapsto s : A \quad \lambda\beta \vdash s = t}{\Gamma \mapsto t : A}$$

*We will write  $\mapsto_{\lambda+\beta}$  for this extended relation.*

*It is a fact, which you may assume (proved in 4A2 of [Hin97]), that it is only ever necessary to use the equality rule once, and at the final stage of a type deduction. Equivalently,  $\Gamma \mapsto_{\lambda+\beta} s : A$  if and only if  $\Gamma \mapsto s' : A$  for some  $s'$  with  $\lambda\beta \vdash s = s'$ .*

**7.4** Show that the set of typable terms in  $TA_{\lambda+\beta}$  is not recursive.

[Hint: Show that Cor. 4.3.3 applies.]

**7.5** Show, by means of a counterexample, that *not* all terms typable in  $TA_{\lambda+\beta}$  are strongly normalizing.

**7.6** Show that all terms typable in  $TA_{\lambda+\beta}$  are weakly normalizing.

## Chapter 8

# Principal Types

---

**Reading:** [Hin97] Ch. 3; [Sel07] 7

---

In this final chapter we address two remaining questions relating to the simple type system.

- (i) We have proved that the typable terms have certain properties. Is the question of whether a term is typable decidable?
- (ii) It seems that typable terms can have many types deduced for them. Can all the types be described systematically?

The answer to both questions is yes, and the proof follows from the correctness of an algorithm which decides whether a term is typable, and finding a “most general” type if it is. The most general types are called **principal types** and all the types of a particular term will be **instances** of the principal type.

Although much of this chapter will be concerned with keeping the technicalities of principal types straight, it can be motivated by only considering a simpler question, of deciding whether a term is typable by looking at the structure of a term. For variables this question is trivial (all variables are typable, and can receive any type) and for abstractions it is very easy ( $\lambda x.s$  is typable if and only if  $s$  is, although the most general type for it will depend on whether  $x$  occurs in  $s$  or not). But the application case is difficult, because to check typability of  $st$  we need to know not only that  $s$  and  $t$  are typable but also that  $s$  can receive an arrow type  $A \Rightarrow B$  and that  $t$  can receive a type which matches  $A$ . The problem of reconciling the most general types of  $s$  and  $t$  in this way is an example of **unification**, an extremely important topic in its own right. Although we focus only on the simple type theory of the pure  $\lambda$ -calculus, the topics of this chapter have direct application to real programming languages.

## 8.1 Type Substitutions and Principal Types

We need to define precisely what a **most general** type is, and what it means for one type to be an **instance** of another. These are technical definitions in terms of substitutions, and the type variables (which previously have not played much part in our reasoning, other than to supply ground types) will become useful.

Just as  $s[t/x]$  indicates a substitution of terms for term variables, we have **type substitutions**, of types for type variables. Unlike terms, we will be concerned with multiple, simultaneous, type substitutions.

**Definition** The notation  $A[T_1/a_1, \dots, T_n/a_n]$  indicates the type  $A$  with each type variable  $a_i$  replaced by the type  $T_i$ . Other type variables are unaffected. Formally,

$$\begin{aligned} a_i[T_1/a_1, \dots, T_n/a_n] &\equiv T_i \\ b[T_1/a_1, \dots, T_n/a_n] &\equiv b, \text{ if } b \text{ is a type variable distinct from all } a_i \\ (A \Rightarrow B)[T_1/a_1, \dots, T_n/a_n] &\equiv A[T_1/a_1, \dots, T_n/a_n] \Rightarrow B[T_1/a_1, \dots, T_n/a_n] \end{aligned}$$

We will need to do a lot of type substitutions, and to avoid having to write out  $[T_1/a_1, \dots, T_n/a_n]$  too often we will give substitutions letters of their own, writing, for example

$$\mathbb{S} \equiv [T_1/a_1, \dots, T_n/a_n]$$

and then  $\mathbb{S}(A)$  for  $A[T_1/a_1, \dots, T_n/a_n]$ . Note that  $\mathbb{S} \equiv [T_1/a_1, \dots, T_n/a_n]$  is only **well-defined** if all the variables  $a_i$  are distinct. (In this respect type substitutions are analogous to type contexts.) In another attempt to save space, we will sometimes write  $[T_1/a_1, \dots, T_n/a_n]$  as  $[\dots, T_i/a_i, \dots]$ , when  $n$  is implied or irrelevant.

Here are some examples of type substitutions:

$$\begin{aligned} (a \Rightarrow b \Rightarrow c)[d/a, (e \Rightarrow e)/c] &\equiv d \Rightarrow b \Rightarrow e \Rightarrow e \\ (a \Rightarrow b \Rightarrow c)[b/a, d/d] &\equiv b \Rightarrow b \Rightarrow c \\ (a \Rightarrow b \Rightarrow c)[(a \Rightarrow c)/a, (b \Rightarrow c)/c] &\equiv (a \Rightarrow c) \Rightarrow b \Rightarrow b \Rightarrow c \end{aligned}$$

Notice that

- (i) It is possible to include trivial substitutions like  $[d/d]$ , which have no effect.
- (ii) It is possible to make “clashing” substitutions, where one type variable is replaced with a type containing another which occurs in the original term.
- (iii) We must be careful to remember the implicit association of  $\Rightarrow$  (to the right).



In each of the examples we just saw, the type  $a \Rightarrow b \Rightarrow c$  is *more general* than the result of the substitution.

**Definition** We say that a type  $B$  is an **instance** of a type  $A$  if there is some type substitution  $\mathbb{S}$  such that  $B \equiv \mathbb{S}(A)$ .

So, for example,  $b \Rightarrow b \Rightarrow b$  is an instance of  $a \Rightarrow b \Rightarrow c$ . So is  $A \Rightarrow B \Rightarrow C$ , for any types  $A$ ,  $B$ , and  $C$ . Now we can make the formal definition of a principal type for a typable term.

**Definition** A **principal type** of a term  $s$  is a type  $A$  such that

- (i)  $\Gamma \vdash s : A$  for some context  $\Gamma$ , and
- (ii) if  $\Gamma' \vdash s : A'$  then  $A'$  is an instance of  $A$ .

If a term has a principal type then it is unique, up to one-one renaming of the type variables in it (this is because any two principal types must be instances of each other).

We now need to extend our notion of type substitution to cover other structure which have types in them.

**Definition** If  $\mathbb{S}$  is a type substitution then it applies to sequences of types, contexts, and deductions, by affecting every type in that structure. Anything else in the structure (e.g. term variables in contexts, terms and term variables in deductions) are unaffected by type substitution.

The concept of **instance** also extends to these structures, similarly.

This allows us to extend the idea of a principal type to a deduction. We will use the symbol  $\Delta$  to range over well-formed deductions in  $\text{TA}_\lambda$ . We will need the following lemma:

**Lemma 8.1.1** If  $\Delta$  is a well-formed deduction then so is  $\mathbb{S}(\Delta)$ , for any type substitution  $\Delta$ .

**Proof** The only difference between  $\Delta$  and  $\mathbb{S}(\Delta)$  are in the types. In particular, the term variables in contexts are unaltered, so the side condition ( $\Gamma$  consistent with  $\Delta$ ) in the application rule cannot be broken by type substitution. Since the same substitution is used throughout, the application and abstraction rules will be preserved. ■

**Definition** A **principal deduction** for  $s$  is a well-formed deduction  $\Delta$  such that

- (i) the conclusion of  $\Delta$  is  $\Gamma \mapsto s : A$ , for some context  $\Gamma$  and type  $A$ , and
- (ii) for any other deduction  $\Delta'$  of  $\Gamma' \mapsto s : A'$ ,  $\Delta'$  is an instance of  $\Delta$ .

For an example, consider the deduction  $\Delta \equiv$

$$\frac{\frac{\overline{\{x:a\} \mapsto x : a}}{\mapsto (\lambda y.x) : b \Rightarrow a}}{\mapsto (\lambda xy.x) : a \Rightarrow b \Rightarrow a}}$$

This is a well-formed deduction. If we take the substitution

$$\mathbb{S} \equiv [(b \Rightarrow a)/a, (a \Rightarrow c)/b]$$

then  $\mathbb{S}(\Delta) \equiv$

$$\frac{\frac{\overline{\{x:b \Rightarrow a\} \mapsto x : b \Rightarrow a}}{\mapsto (\lambda y.x) : (a \Rightarrow c) \Rightarrow b \Rightarrow a}}{\mapsto (\lambda xy.x) : (b \Rightarrow a) \Rightarrow (a \Rightarrow c) \Rightarrow b \Rightarrow a}}$$

If we apply the Subject Construction Lemma to  $\lambda xy.x$ , we can see that in fact *every* deduction of  $\Gamma \mapsto \lambda xy.x$  must be of the shape of  $\Delta$ , possibly with other types replacing the type variables  $a$  and  $b$  throughout. This is precisely what it means for  $\Delta$  to be the principal deduction of  $\lambda xy.x$ , and it implies that the principal type of  $\lambda xy.x$  is  $a \Rightarrow b \Rightarrow a$ .

## 8.2 Lemmas about Type Substitutions

To reason precisely about principal types and principal deductions, we need to reason about instances. And to do this, we need some rather messy algebraic properties of substitutions. This seems unavoidable.

First, some definitions relating to type substitutions.

### Definition

- (i) If  $\mathbb{S}$  is the substitution  $[A_1/a_1, \dots, A_m/a_m]$  we say that each  $[A_i/a_i]$  is a **component** of  $\mathbb{S}$ .
- (ii) A component is **trivial** if it is of the form  $[a/a]$  for some type variable  $a$ .
- (iii) We write  $\mathbb{E}$  for the **empty substitution** which does nothing, so  $\mathbb{E}(A) \equiv A$  for all types  $A$ .

(iv) We use the symbol  $\equiv$  for exact identity of substitutions, i.e.

$$[A_1/a_1, \dots, A_m/a_m] \equiv [B_1/b_1, \dots, B_n/b_n]$$

only if  $m = n$ , and all the components  $[A_i/a_i]$  correspond bijectively with the  $[B_j/b_j]$ , with  $a_i \equiv b_j$  and  $A_i \equiv B_j$ . But the order of the components in a substitution is irrelevant. A weaker notion of equality is **extensional equality**: we write  $\mathbb{S} =_{\text{ext}} \mathbb{T}$  if  $\mathbb{S}(A) \equiv \mathbb{T}(A)$  for all types  $A$ . (This is weaker because it allows  $\mathbb{S}$  and  $\mathbb{T}$  to differ in trivial components.)

(v) If  $A$  is a type then  $\text{Vars}(A)$  is the set of all type variables occurring in  $A$ . We extend this to deductions in the natural way:  $\text{Vars}(\Delta)$  is the set of all type variables occurring in any of the types or contexts anywhere in  $\Delta$ .

(vi) If  $\mathbb{S} \equiv [A_1/a_1, \dots, A_n/a_n]$  then the **variable-domain** of  $\mathbb{S}$  is

$$\text{Dom}(\mathbb{S}) = \{a_1, \dots, a_n\}$$

and the **variable-range** of  $\mathbb{S}$  is

$$\text{Range}(\mathbb{S}) = \text{Vars}(A_1) \cup \dots \cup \text{Vars}(A_n).$$

Now we define two operations on type substitutions. Later, we will use them to build up complex type substitutions, starting from the empty substitution and combining them until a substitution with some required properties has been found.

### Definition

**Union** of substitutions: Suppose

$$\begin{aligned} \mathbb{S} &\equiv [A_1/a_1, \dots, A_m/a_m] \text{ and} \\ \mathbb{T} &\equiv [B_1/b_1, \dots, B_n/b_n], \end{aligned}$$

with all  $a_i$  distinct from all  $b_j$ . We define

$$\mathbb{S} \cup \mathbb{T} = [A_1/a_1, \dots, A_m/a_m, B_1/b_1, \dots, B_n/b_n].$$

It also makes sense to form  $\mathbb{S} \cup \mathbb{T}$  when some  $a_i \equiv b_j$ , as long as  $A_i \equiv B_j$  in such situations.

**Composition** of substitutions: Suppose

$$\begin{aligned} \mathbb{S} &\equiv [A_1/a_1, \dots, A_m/a_m, C_1/c_1, \dots, C_n/c_n] \text{ and} \\ \mathbb{T} &\equiv [B_1/b_1, \dots, B_p/b_p, D_1/d_1, \dots, D_n/d_n], \end{aligned}$$

with all  $a_i, b_j, c_k$  distinct (so that we have explicitly identified  $\text{Dom}(\mathbb{S}) \cap \text{Dom}(\mathbb{T})$  – note that possibly some  $C_i$  and  $D_i$  are different in which case  $\mathbb{S} \cup \mathbb{T}$  would not be well-defined). We define

$$\mathbb{S} \circ \mathbb{T} = [\dots, A_i/a_i, \dots, \mathbb{S}(B_i)/b_i, \dots, \mathbb{S}(D_i)/d_i, \dots].$$

Union is intuitively obvious. Composition is necessary when we want to build a single substitution with the effect of performing two substitutions in series (note that the order matters so that  $\circ$  is not commutative). The operations have the following properties:

**Lemma 8.2.1** For any substitutions  $\mathbb{S}$  and  $\mathbb{T}$ ,

(i) for any type  $A$ ,

$$(\mathbb{S} \circ \mathbb{T})(A) = \mathbb{S}(\mathbb{T}(A)).$$

(ii)  $Dom(\mathbb{S} \cup \mathbb{T}) = Dom(\mathbb{S}) \cup Dom(\mathbb{T})$ .

(iii)  $Dom(\mathbb{S} \circ \mathbb{T}) = Dom(\mathbb{S}) \cup Dom(\mathbb{T})$ .

**Proof** (i) Let  $\mathbb{S}$  and  $\mathbb{T}$  be as in the definition of  $\circ$ . Then

$$\begin{aligned} \mathbb{S}(\mathbb{T}(a_i)) &= \mathbb{S}(a_i) = A_i = (\mathbb{S} \circ \mathbb{T})(a_i) \\ \mathbb{S}(\mathbb{T}(b_i)) &= \mathbb{S}(B_i) = (\mathbb{S} \circ \mathbb{T})(b_i) \\ \mathbb{S}(\mathbb{T}(c_i)) &= \mathbb{S}(D_i) = (\mathbb{S} \circ \mathbb{T})(c_i) \end{aligned}$$

and the rest follows because all substitutions pull through the  $\Rightarrow$ .

(ii) and (iii) Immediate from the definition. ■

A brief example: if  $\mathbb{S} \equiv [(a \Rightarrow b)/a, (b \Rightarrow a)/b]$  and  $\mathbb{T} \equiv [(a \Rightarrow b \Rightarrow c)/a, d/c]$  then  $\mathbb{S} \cup \mathbb{T}$  is not well-defined, but

$$\mathbb{S} \circ \mathbb{T} \equiv [(a \Rightarrow b) \Rightarrow (b \Rightarrow a) \Rightarrow c)/a, (b \Rightarrow a)/b, d/c]$$

and

$$\mathbb{T} \circ \mathbb{S} \equiv [(a \Rightarrow b \Rightarrow c) \Rightarrow a)/a, (b \Rightarrow (a \Rightarrow b \Rightarrow c))/b, d/c].$$

Finally, we need a rather technical lemma which states conditions under which a composition  $\mathbb{S} \circ \mathbb{T}$  can be “extended” by taking its union with  $\mathbb{R}$ , and the whole substitution expressed as a single composition (this will be necessary in the proof of correctness of the Principal Type Algorithm).

**Lemma 8.2.2 (Composition-Extension)** Let  $\mathbb{R}$ ,  $\mathbb{S}$  and  $\mathbb{T}$  substitutions satisfying

(i)  $Dom(\mathbb{R}) \cap (Dom(\mathbb{S}) \cup Dom(\mathbb{T})) = \emptyset$ , and

(ii)  $Dom(\mathbb{R}) \cap Range(\mathbb{T}) = \emptyset$ .

Then

$$\mathbb{R} \cup (\mathbb{S} \circ \mathbb{T}) \equiv (\mathbb{R} \cup \mathbb{S}) \circ \mathbb{T}$$

and both sides are well-defined substitutions.

**Proof** Directly from (i) and Lemma 8.2.1(iii),  $Dom(\mathbb{R}) \cap Dom(\mathbb{S} \circ \mathbb{T}) = \emptyset$

so  $\mathbb{R} \cup (\mathbb{S} \circ \mathbb{T})$  is well-defined. Also, (i) implies that  $Dom(\mathbb{R}) \cap Dom(\mathbb{S}) = \emptyset$ , so  $(\mathbb{R} \cup \mathbb{S}) \circ \mathbb{T}$  is also well-defined.

By (i), the type variables affected by  $\mathbb{R}$  are disjoint from those affected by  $\mathbb{S}$  or  $\mathbb{T}$ . So let us write out

$$\begin{aligned}\mathbb{R} &\equiv [R_1/r_1, \dots, R_l/r_l] \\ \mathbb{S} &\equiv [A_1/a_1, \dots, A_m/a_m, C_1/c_1, \dots, C_n/c_n] \\ \mathbb{T} &\equiv [B_1/b_1, \dots, B_p/b_p, D_1/d_1, \dots, D_n/d_n]\end{aligned}$$

where the sets of type variables  $a_1, \dots, a_m$ ,  $b_1, \dots, b_p$ ,  $c_1, \dots, c_n$ ,  $r_1, \dots, r_l$  are all disjoint.

From the definitions of  $\cup$  and  $\circ$ ,

$$\mathbb{R} \cup (\mathbb{S} \circ \mathbb{T}) \equiv [\dots, R_i/r_i, \dots, A_i/a_i, \dots, \mathbb{S}(B_i)/b_i, \dots, \mathbb{S}(D_i)/d_i, \dots].$$

On the other hand,

$$\begin{aligned}(\mathbb{R} \cup \mathbb{S}) \circ \mathbb{T} &\equiv [\dots, R_i/r_i, \dots, A_i/a_i, \dots, C_i/c_i, \dots] \circ [\dots, B_i/b_i, \dots, D_i/d_i, \dots] \\ &\equiv [\dots, R_i/r_i, \dots, A_i/a_i, \dots, (\mathbb{R} \cup \mathbb{S})(B_i)/b_i, \dots, (\mathbb{R} \cup \mathbb{S})(D_i)/d_i, \dots] \\ &\equiv [\dots, R_i/r_i, \dots, A_i/a_i, \dots, \mathbb{S}(B_i)/b_i, \dots, \mathbb{S}(D_i)/d_i, \dots]\end{aligned}$$

the second equation by the definition of composition, and the third because  $(\mathbb{R} \cup \mathbb{S})(B_i) = \mathbb{S}(B_i)$  and  $(\mathbb{R} \cup \mathbb{S})(D_i) = \mathbb{S}(D_i)$ , which follows from (ii). ■

## 8.3 Unification

We are now in a position to set about reconciling types.

**Definition** Let  $A$  and  $B$  be types. A substitution  $\mathbb{U}$  which satisfies

$$\mathbb{U}(A) \equiv \mathbb{U}(B)$$

is called a **unifier** for the pair  $(A, B)$ . If a unifier exists we say that  $A$  and  $B$  are **unifiable**, and the common type  $\mathbb{U}(A) \equiv \mathbb{U}(B)$  is called the **unification** (or **unifying type**) of  $A$  and  $B$ .

A unifier is a special case of a related concept: we say that  $A$  and  $B$  have a **common instance** if there are substitutions  $\mathbb{S}$  and  $\mathbb{T}$  such that  $\mathbb{S}(A) \equiv \mathbb{T}(B)$ . Every unifiable pair has a common instance (set  $\mathbb{S} \equiv \mathbb{T}$ ) and it seems that unifiers in general will be harder to find than common instances. But in the case when  $Vars(A) \cap Vars(B) = \emptyset$  then the existence of a common instance implies the existence of a unifier.

Finding whether two types have a unifier, a common instance, or solving the similar problem of **matching** (finding a substitution  $\mathbb{S}$  such that  $\mathbb{S}(A) \equiv B$ ) are ubiquitous issues in computer science, and they apply to all sorts of other recursive structures, as well as types. Whether the questions are decidable depends very much on the data structure under consideration. Sadly, we do not have time to study this fascinating topic further, and indeed we will have to take a very cursory approach to unification of types.

Here are some examples of unification. Let  $A \equiv a \Rightarrow b$  and  $B \equiv b \Rightarrow a$ . A unifier for  $(A, B)$  is  $\mathbb{S} \equiv [a/b]$  (another is  $\mathbb{S}' \equiv [b/a]$ , and yet another is  $\mathbb{S}'' \equiv [C/a, C/b]$ , where  $C$  is any type). For a more difficult example, let  $A \equiv a \Rightarrow b \Rightarrow b$  and  $B \equiv (c \Rightarrow c) \Rightarrow d$ . A unifier is

$$\mathbb{S} \equiv [(e \Rightarrow e)/a, e/b, e/c, (e \Rightarrow e)/d],$$

and the unification is  $(e \Rightarrow e) \Rightarrow (e \Rightarrow e)$ .

Finally, if  $A \equiv a \Rightarrow a \Rightarrow a$  and  $B \equiv b \Rightarrow b$  then  $A$  and  $B$  are not unifiable.

We will be particularly interested in “simplest” unifiers, where simplest is taken to mean most general.

**Definition** A **most general unifier** (usually written **m.g.u.**) for a pair of types  $(A, B)$  is a substitution  $\mathbb{U}$  such that

$$\mathbb{U}(A) \equiv \mathbb{U}(B)$$

and, for all other substitutions  $\mathbb{S}$  with  $\mathbb{S}(A) \equiv \mathbb{S}(B)$  we have

$$\mathbb{S}(A) \equiv \mathbb{S}'(\mathbb{U}(A))$$

for some substitution  $\mathbb{S}'$ . When such a  $\mathbb{U}$  exists we say that the common type  $\mathbb{U}(A) \equiv \mathbb{U}(B)$  is the **most general unification** of  $A$  and  $B$ .

We return to the previous example. If  $A \equiv a \Rightarrow b$  and  $B \equiv b \Rightarrow a$  then either  $\mathbb{S} \equiv [a/b]$  or  $\mathbb{S}' \equiv [b/a]$  are most general unifiers, and so is  $\mathbb{S}'' \equiv [c/a, c/b]$  where  $c$  is any type variable, but  $\mathbb{T} \equiv [C/a, C/b]$  is not most general when  $C$  is something other than a single type variable.

If  $A \equiv a \Rightarrow b \Rightarrow b$  and  $B \equiv (c \Rightarrow c) \Rightarrow d$ , the most general unifier is

$$\mathbb{S} \equiv [(e \Rightarrow e)/a, f/b, e/c, (f \Rightarrow f)/d],$$

and the most general unification is  $(e \Rightarrow e) \Rightarrow (f \Rightarrow f)$ .

As we have seen, m.g.u.’s are not unique, but they *are* unique up to renaming of type variables (for a proof see [Hin97, 3D2]). Because we can always rename type variables without affecting whether one type is an instance of another (as long as distinct variables are renamed to distinct variables), we have:

**Lemma 8.3.1** Let  $\mathcal{V}$  be any finite set of type variables. Then a pair of types  $(A, B)$  has a m.g.u. if and only if it has an m.g.u  $\mathbb{U}$  satisfying

- (i)  $Dom(\mathbb{U}) = Vars(A) \cup Vars(B)$ ,
- (ii)  $Range(\mathbb{U}) \cap \mathcal{V} = \emptyset$ .

**Proof** We can ensure this by

- (i) deleting any components  $[C_i/c_i]$  where  $c_i \notin Vars(A) \cup Vars(B)$  (they don't affect  $\mathbb{U}(A)$  or  $\mathbb{U}(B)$ );
- (ii) adding any trivial components  $[a_i/a_i]$  where required, to pad the domain of  $\mathbb{U}$ ;
- (iii) renaming the type variables in  $\mathbb{U}$  to avoid conflict with  $\mathcal{V}$ .

■

Recall that we extend the definition of type substitution to sequences of types, by applying the substitution to each type in the sequence. In a similar way, we extend unification to pairs of sequences of types:

**Definition** Two type sequences of equal length  $\langle A_1, \dots, A_n \rangle$  and  $\langle B_1, \dots, B_n \rangle$  are **unifiable** if there is a substitution  $\mathbb{U}$  such that

$$\mathbb{U}(A_i) = \mathbb{U}(B_i), \text{ for all } i$$

and in this case  $\mathbb{U}$  is called the **unifier** for the two sequences.

Most general unifiers of sequences are defined analogously.

The problem of finding a (most general) unifier for two sequences of types can, however, be reduced to the problem of finding a (most general) unifier for two individual types: given  $\langle A_1, \dots, A_n \rangle$  and  $\langle B_1, \dots, B_n \rangle$  pick a fresh type variable  $c$  and construct new types

$$A' \equiv A_1 \Rightarrow A_2 \Rightarrow \dots \Rightarrow A_n \Rightarrow c \text{ and } B' \equiv B_1 \Rightarrow B_2 \Rightarrow \dots \Rightarrow B_n \Rightarrow c.$$

It is simple to show that any unifier for  $A'$  and  $B'$  is a unifier for the sequences  $\langle A_1, \dots, A_n \rangle$  and  $\langle B_1, \dots, B_n \rangle$ , and that the most general unifier for  $\langle A_1, \dots, A_n \rangle$  and  $\langle B_1, \dots, B_n \rangle$  can be obtained from the most general unifier for  $A'$  and  $B'$  by throwing out the component  $[C/c]$  (if there is one).

We are now ready to describe an algorithm for finding most general unifiers. The following is due to Robinson [Rob65], and was the first to have a fully worked-through proof of correctness. Many other unification algorithms were used, from the 1960s onwards, and Robinson's is by no means the most efficient (although it is probably the simplest to understand). Unfortunately, we cannot include the correctness proof in this course.

**Theorem 8.3.2 (Robinson’s Unification Algorithm)** If a pair of types  $(A, B)$  has a unifier then it has a m.g.u. Furthermore, there is an algorithm which determines whether a pair is unifiable and constructs an m.g.u. when they are. The algorithm is as follows. Input types  $A$  and  $B$ .

1. Initially, set  $\mathbb{U} \equiv \mathbb{E}$ .
2. Compute  $\mathbb{U}(A)$  and  $\mathbb{U}(B)$ . If they are identical, terminate the computation and declare that  $A$  and  $B$  are unifiable, with  $\mathbb{U}$  their m.g.u.
3. If  $\mathbb{U}(A)$  and  $\mathbb{U}(B)$  are not identical, they must be of the form (when written as strings):

$$\begin{array}{l} \text{string}_1 \ c \ \text{string}_2 \\ \text{string}_1 \ C \ \text{string}_3 \end{array}$$

where the two strings  $\text{string}_1$  are identical, and  $c$  is some type variable, and  $C$  is some type not identical to the single type variable  $c$ . (In other words, we find the first place where  $\mathbb{U}(A)$  and  $\mathbb{U}(B)$  differ: it is not difficult to prove that, at this position at least one of the two is a type variable.)

4. List  $\text{Vars}(C)$ . If  $c \in \text{Vars}(C)$  terminate the computation, stating that  $A$  and  $B$  are not unifiable.
5. Otherwise, replace  $\mathbb{U}$  by  $[C/c] \circ \mathbb{U}$  and return to step 2.

The proof that this algorithm is correct (which is certainly not immediately obvious) can be found in [Rob65].

## 8.4 The Principal Type Algorithm

We can now present a principal type algorithm, which decides whether a given term is typable and computes a principal type if it is. The algorithm here follows that in 3E of [Hin97] very closely, which is due to Hindley and was presented in the late 1960s; it is one of the first, although similar ideas are implicit in Curry’s 1950s work. Once functional programming took off, the topic became much more widely studied, and there is now a lot of literature on such algorithms, including more efficient ones than this, but they are generally much less easy to describe. (Modern functional languages such as ML use highly-optimized type inference algorithms.)

We have already mentioned that principal types for variables and abstractions are quite easy to compute. For the case of applications, the unification algorithm is key, allowing us to compute a principal type for an application  $pq$  from the principal types for  $p$  and  $q$ . (If the principal type of  $p$  is  $A$ , and



the principal type of  $q$  is  $B$ , we will find a most general unifier  $\mathbb{U}$  for the pair  $(A, B \Rightarrow C)$ : the principal type of  $pq$  will be  $\mathbb{U}(C)$ .)

In the following presentation we will interleave the correctness proof, written in italics, with the statement of the algorithm.

### Algorithm

The input is a term of the untyped  $\lambda$ -calculus,  $s$ . The output is either a correct statement that  $s$  is not typable, or a principal deduction for  $s$ .

(Throughout, we will write  $\Delta_s$  to mean a principal deduction for the term  $s$ .)

#### The Variable Case

If  $s \equiv x$ , choose any type variable  $a$  and set  $\Delta_s$  to be the deduction

$$\frac{}{x:a \mapsto x : a}$$

*Justification of the variable case: Obvious.*

#### The Abstraction Case

If  $s \equiv \lambda x.p$  then there are two subcases, which we treat separately.

**Abstraction case (a):**  $x$  does occur free in  $p$ . Suppose that the free variables of  $p$  are exactly  $\{x, y_1, \dots, y_n\}$ .

Apply this algorithm recursively to  $p$ . If  $p$  is not typable, terminate the computation, with output that  $s$  is not typable. If  $p$  has principal deduction  $\Delta_p$  then its conclusion must be of the form

$$\{x:A, y_1:B_1, \dots, y_n:B_n\} \mapsto p : C$$

for some type  $C$ . Applying the rule (abs-main) to this, we create a deduction  $\Delta_{\lambda x.p}$  with conclusion

$$\{y_1:B_1, \dots, y_n:B_n\} \mapsto \lambda x.p : A \Rightarrow C$$

Return  $\Delta_{\lambda x.p}$  as the principal deduction of  $s$ .

*Justification of abstraction case (a): That the untypability of  $p$  implies the untypability of  $s$  is immediate from the Subject Construction Lemma. So we must show that  $\Delta_{\lambda x.p}$  is principal for  $s$ . Let  $\Delta$  be any other deduction whose*

conclusion is the term  $\lambda x.p$ . By the Subject Construction Lemma, the last step in  $\Delta$  must be the (abs-main) rule, of the form

$$\frac{\{x:A', y_1:B'_1, \dots, y_n:B'_n\} \mapsto p : C'}{\{y_1:B'_1, \dots, y_n:B'_n\} \mapsto \lambda x.p : A' \Rightarrow C'}$$

for some types  $A', B'_1, \dots, B'_n, C'$ . But the sub-deduction  $\Delta'$  obtained from  $\Delta$  by removing the last step is a deduction which assigns a type to  $p$ , and  $\Delta_p$  is the principal deduction of  $p$ . Therefore  $\Delta'$  must be an instance of  $\Delta_p$ , which means that there is a type substitution  $\mathbb{S}$  with  $\Delta' \equiv \mathbb{S}(\Delta_p)$ . In particular,

$$A' \equiv \mathbb{S}(A), \quad B'_i \equiv \mathbb{S}(B_i), \text{ for each } i, \quad C' \equiv \mathbb{S}(C)$$

which means that  $\Delta = \mathbb{S}(\Delta_{\lambda x.p})$ . We have proved that  $\Delta$  must be an instance of  $\Delta_{\lambda x.p}$ , and therefore that  $\Delta_{\lambda x.p}$  is a principal deduction of  $\lambda x.p$ .

**Abstraction case (b):**  $x$  does not occur free in  $p$ . Suppose that the free variables of  $p$  are exactly  $\{y_1, \dots, y_n\}$ .

Apply this algorithm recursively to  $p$ . If  $p$  is not typable, terminate the computation, with output that  $s$  is not typable. If  $p$  has principal deduction  $\Delta_p$  then its conclusion must be of the form

$$\{y_1:B_1, \dots, y_n:B_n\} \mapsto p : C$$

for some type  $C$ . Pick some fresh type variable  $d$  not occurring anywhere in  $\Delta_p$  and apply the rule (abs-vac) to create a deduction  $\Delta_{\lambda x.p}$  with conclusion

$$\{y_1:B_1, \dots, y_n:B_n\} \mapsto \lambda x.p : d \Rightarrow C$$

Return  $\Delta_{\lambda x.p}$  as the principal deduction of  $s$ .

*Justification of abstraction case (b):* Similar to case (a). Let  $\Delta$  be any other deduction whose conclusion is the term  $\lambda x.p$ . By the Subject Construction Lemma, the last step in  $\Delta$  must be the (abs-vac) rule, of the form

$$\frac{\{y_1:B'_1, \dots, y_n:B'_n\} \mapsto p : C'}{\{y_1:B'_1, \dots, y_n:B'_n\} \mapsto \lambda x.p : A' \Rightarrow C'}$$

for some types  $A', B'_1, \dots, B'_n, C'$ . But the sub-deduction  $\Delta'$  obtained from  $\Delta$  by removing the last step is a deduction which assigns a type to  $p$ , and  $\Delta_p$  is the principal deduction of  $p$ . Therefore  $\Delta'$  must be an instance of  $\Delta_p$ , which means that there is a type substitution  $\mathbb{S}$  with

$$B'_i \equiv \mathbb{S}(B_i), \text{ for each } i, \quad C' \equiv \mathbb{S}(C).$$

Now set  $\mathbb{S}' \equiv \mathbb{S} \cup [A'/d]$ , well-defined because  $d$  is a fresh type variable. We have  $\Delta = \mathbb{S}'(\Delta_{\lambda x.p})$  and therefore  $\Delta_{\lambda x.p}$  is a principal deduction of  $\lambda x.p$ .

### The Application Case

If  $s \equiv pq$  then there are a number of subcases, into which we split as the algorithm progresses.

First, apply the algorithm recursively to  $p$  and  $q$ . If either  $p$  or  $q$  is untypable, terminate the computation with output that  $s$  is untypable.

*Justification: every subterm of a typable term must be typable, by the Subject Construction Lemma.*

Let  $\Delta_p$  and  $\Delta_q$  be principal deductions for  $p$  and  $q$ , renaming if necessary so that these two deductions contain no type variables in common. Suppose that

$$\begin{aligned} \text{FV}(p) &= \{x_1, \dots, x_l, z_1, \dots, z_n\} \\ \text{FV}(q) &= \{y_1, \dots, y_m, z_1, \dots, z_n\} \end{aligned}$$

(so that we have identified the common free variables). Now we split into two subcases (a) and (b), depending on the principal type for  $p$ .

**Application case (a):** The principal type of  $p$  is an arrow type. Suppose that this type is  $A \Rightarrow B$ , and that the principal type of  $q$  is  $C$ . The conclusions of  $\Delta_p$  and  $\Delta_q$  must be of the form

$$\begin{aligned} \{x_1:A_1, \dots, x_l:A_l, z_1:C_1, \dots, z_n:C_n\} &\mapsto p : A \Rightarrow B \\ \{y_1:B_1, \dots, y_m:B_m, z_1:D_1, \dots, z_n:D_n\} &\mapsto q : C \end{aligned}$$

Apply the unification algorithm to the pair of sequences of types

$$\langle C_1, \dots, C_n, A \rangle \text{ and } \langle D_1, \dots, D_n, C \rangle.$$

**Application subcase (a1):**  $\langle C_1, \dots, C_n, A \rangle$  and  $\langle D_1, \dots, D_n, C \rangle$  no unifier. Then terminate the computation, with output that  $s$  is untypable.

*Justification of application subcase (a1): We will prove that, if  $s \equiv pq$  were typable, then the two sequences  $\langle C_1, \dots, C_n, A \rangle$  and  $\langle D_1, \dots, D_n, C \rangle$  would have some unifier.*

If  $s \equiv pq$  is typable, then there is a deduction  $\Delta$  with conclusion

$$\{x_1:E_1, \dots, x_l:E_l, y_1:F_1, \dots, y_m:F_m, z_1:G_1, \dots, z_n:G_n\} \mapsto pq : H$$

The Subject Construction Lemma tells us that the last step of this deduction was the application rule, applied to deductions  $\Delta_1$  and  $\Delta_2$  with conclusions

$$\begin{aligned} \{x_1:E_1, \dots, x_l:E_l, z_1:G_1, \dots, z_n:G_n\} &\mapsto p : I \Rightarrow H \\ \{y_1:F_1, \dots, y_m:F_m, z_1:G_1, \dots, z_n:G_n\} &\mapsto q : I \end{aligned}$$

for some type  $I$ . But we have that  $\Delta_p$  is a principal deduction for  $p$ , and  $\Delta_q$  a principal deduction for  $q$ , so there are type substitutions  $\mathbb{R}_1$  and  $\mathbb{R}_2$  such that

$$\Delta_1 \equiv \mathbb{R}_1(\Delta_p) \text{ and } \Delta_2 \equiv \mathbb{R}_2(\Delta_q).$$

We may also assume that  $\text{Dom}(\mathbb{R}_1) = \text{Vars}(\Delta_p)$  and  $\text{Dom}(\mathbb{R}_2) = \text{Vars}(\Delta_q)$ , and we have assumed that these sets of type variables are disjoint, so that

$$\mathbb{R} \equiv \mathbb{R}_1 \cup \mathbb{R}_2$$

is well-defined. Now we have that  $\Delta_1 \equiv \mathbb{R}(\Delta_p)$  and  $\Delta_2 \equiv \mathbb{R}(\Delta_q)$ . Comparing the types in the conclusions of  $\Delta_1$  and  $\mathbb{R}(\Delta_p)$  we have that

$$G_i \equiv \mathbb{R}(C_i) \text{ for each } i, \text{ and } I \equiv A$$

and doing the same for  $\Delta_2$  and  $\mathbb{R}(\Delta_q)$  we have that

$$G_i \equiv \mathbb{R}(D_i) \text{ for each } i, \text{ and } I \equiv C.$$

This is sufficient to show that  $\mathbb{R}$  is a unifier for  $\langle C_1, \dots, C_n, A \rangle$  and  $\langle D_1, \dots, D_n, C \rangle$ .

**Application subcase (a2):**  $\langle C_1, \dots, C_n, A \rangle$  and  $\langle D_1, \dots, D_n, C \rangle$  have a most general unifier  $\mathbb{U}$ . Apply Lemma 8.3.1 to ensure that

- (i)  $\text{Dom}(\mathbb{U}) = \text{Vars}(C_1, \dots, C_n, A, D_1, \dots, D_n, C)$ , and
- (ii)  $\text{Range}(\mathbb{U}) \cap \mathcal{V} = \emptyset$ .

where  $\mathcal{V} = (\text{Vars}(\Delta_p) \cup \text{Vars}(\Delta_q)) - \text{Dom}(\mathbb{U})$ .

Apply  $\mathbb{U}$  to  $\Delta_p$  and  $\Delta_q$ : their conclusions become

$$\begin{aligned} \{x_1:A'_1, \dots, x_l:A'_l, z_1:C'_1, \dots, z_n:C'_n\} &\mapsto p : A' \Rightarrow B' \\ \{y_1:B'_1, \dots, y_m:B'_m, z_1:D'_1, \dots, z_n:D'_n\} &\mapsto q : C' \end{aligned}$$

where each  $X'$  denotes  $\mathbb{U}(X)$ . Because  $\mathbb{U}$  is a unifier for  $\langle C_1, \dots, C_n, A \rangle$  and  $\langle D_1, \dots, D_n, C \rangle$ , we can use the application rule to make a combined deduction  $\Delta_{pq}$ , whose conclusion is

$$\{x_1:A'_1, \dots, x_l:A'_l, y_1:B'_1, \dots, y_m:B'_m, z_1:C'_1, \dots, z_n:C'_n\} \mapsto pq : B'$$

This is the principal deduction of  $s$ .

*Justification of application subcase (a2):*  $\Delta_{pq}$  is clearly a well-formed deduction. So we must prove that any other deduction  $\Delta$  which concludes a type for  $s \equiv pq$  is an instance of  $\Delta_{pq}$ .

Define  $\Delta_1$ ,  $\Delta_2$ , and  $\mathbb{R}$  just as in the justification of subcase (a1) above. Now we have that  $\text{Dom}(\mathbb{R}) = \text{Vars}(\Delta_p) \cup \text{Vars}(\Delta_q)$ ; by the definition of  $\mathcal{V}$  we can split  $\mathbb{R}$  into two parts:

$$\mathbb{R} \equiv \mathbb{R}' \cup \mathbb{R}''$$

where  $\text{Dom}(\mathbb{R}') \subseteq \mathcal{V}$  and  $\text{Dom}(\mathbb{R}'') \subseteq \text{Dom}(\mathbb{U})$ .

But consider  $\mathbb{R}''$ : since all of  $C_1, \dots, C_n, A, D_1, \dots, D_n, C \notin \mathcal{V}$  we have that  $\mathbb{R}''$  must be a unifier for  $\langle C_1, \dots, C_n, A \rangle$  and  $\langle D_1, \dots, D_n, C \rangle$  (as in the

justification of subcase (a1)). But  $\mathbb{U}$  is the most general unifier, so there is a substitution  $\mathbb{S}$  such that

$$\mathbb{R}'' =_{\text{ext}} \mathbb{S} \circ \mathbb{U}.$$

And, since  $\mathbb{R} \equiv \mathbb{R}' \cup \mathbb{R}''$ , we have

$$\mathbb{R} =_{\text{ext}} \mathbb{R}' \cup (\mathbb{S} \circ \mathbb{U}).$$

Now it is routine to check (using (ii) and the definition of  $\mathbb{R}'$ ) that the hypotheses of the Composition-Extension Lemma 8.2.2 are satisfied, which guarantees

$$\mathbb{R} =_{\text{ext}} (\mathbb{R}' \cup \mathbb{S}) \circ \mathbb{U}.$$

Therefore, just as we had  $\Delta_1 \equiv \mathbb{R}(\Delta_p)$  and  $\Delta_2 \equiv \mathbb{R}(\Delta_q)$  in the justification of subcase (a1), we now have

$$\Delta_1 \equiv (\mathbb{R}' \cup \mathbb{S})(\mathbb{U}(\Delta_p)) \text{ and } \Delta_2 \equiv (\mathbb{R}' \cup \mathbb{S})(\mathbb{U}(\Delta_q)).$$

This gives that

$$\Delta \equiv (\mathbb{R}' \cup \mathbb{S})(\Delta_{pq})$$

implying that  $\Delta_{pq}$  is a principal deduction.

**Application case (b):** The principal type of  $p$  is atomic. Suppose it is the type variable  $a$ , and that the principal type of  $q$  is  $C$ . The conclusions of  $\Delta_p$  and  $\Delta_q$  must be of the form

$$\begin{aligned} \{x_1:A_1, \dots, x_l:A_l, z_1:C_1, \dots, z_n:C_n\} &\mapsto p : a \\ \{y_1:B_1, \dots, y_m:B_m, z_1:D_1, \dots, z_n:D_n\} &\mapsto q : C \end{aligned}$$

Pick a fresh type variable  $b$  and apply the unification algorithm to the pair of sequences of types

$$\langle C_1, \dots, C_n, a \rangle \text{ and } \langle D_1, \dots, D_n, C \Rightarrow b \rangle.$$

**Application subcase (b1):**  $\langle C_1, \dots, C_n, a \rangle$  and  $\langle D_1, \dots, D_n, C \Rightarrow b \rangle$  no unifier. Then terminate the computation, with output that  $s$  is untypable.

*Justification of application subcase (b1):* Almost identical to the justification of (a1); suppose that  $s$  is typable and construct a unifier for  $\langle C_1, \dots, C_n, a \rangle$  and  $\langle D_1, \dots, D_n, C \Rightarrow b \rangle$ .

The proof is exactly the same up until the construction of the substitution  $\mathbb{R}$ , which in this case becomes

$$\mathbb{R} \equiv \mathbb{R}_1 \cup \mathbb{R}_2 \cup [H/b]$$

which is well-defined for the same reason as before, and also because  $b$  was a fresh type variable. Now it is straightforward to prove that  $\mathbb{R}$  is a unifier for  $\langle C_1, \dots, C_n, a \rangle$  and  $\langle D_1, \dots, D_n, C \Rightarrow b \rangle$ , completing the justification.

**Application subcase (b2):**  $\langle C_1, \dots, C_n, a \rangle$  and  $\langle D_1, \dots, D_n, C \Rightarrow b \rangle$  have a most general unifier  $\mathbb{U}$ . Apply Lemma 8.3.1 to ensure that

- (i)  $Dom(\mathbb{U}) = Vars(C_1, \dots, C_l, a, D_1, \dots, D_n, C \Rightarrow b)$ , and
- (ii)  $Range(\mathbb{U}) \cap \mathcal{V} = \emptyset$ .

where  $\mathcal{V} = (Vars(\Delta_p) \cup Vars(\Delta_q)) - Dom(\mathbb{U})$ .

Apply  $\mathbb{U}$  to  $\Delta_p$  and  $\Delta_q$ : their conclusions become

$$\begin{aligned} \{x_1:A'_1, \dots, x_l:A'_l, z_1:C'_1, \dots, z_n:C'_n\} &\mapsto p : C' \Rightarrow b' \\ \{y_1:B'_1, \dots, y_m:B'_m, z_1:D'_1, \dots, z_n:D'_n\} &\mapsto q : C' \end{aligned}$$

where each  $X'$  denotes  $\mathbb{U}(X)$ . Because  $\mathbb{U}$  is a unifier for  $\langle C_1, \dots, C_n, a \rangle$  and  $\langle D_1, \dots, D_n, C \Rightarrow b \rangle$ , we can use the application rule to make a combined deduction  $\Delta_{pq}$ , whose conclusion is

$$\{x_1:A'_1, \dots, x_l:A'_l, y_1:B'_1, \dots, y_m:B'_m, z_1:C'_1, \dots, z_n:C'_n\} \mapsto pq : b'$$

This is the principal deduction of  $s$ .

*Justification of application subcase (b2): Identical to the justification of subcase (b2), with a few of the types altered appropriately.*

Finally, having proved correctness of each step of this algorithm, we should also prove that it terminates. Thankfully, this is immediate: each recursive step of the algorithm is applied to a shorter term than the input.

## 8.5 Consequences of the Principal Type Algorithm

The immediate consequence is that certain questions about  $TA_\lambda$  are decidable. If we wanted to build a functional programming language on top of  $TA_\lambda$ , the type inference question had better be decidable, if we want the compiler to recognise type errors for us!

**Corollary 8.5.1** The following questions are decidable:

- (i) Given a term  $s$  and a type  $A$ , does  $\Gamma \vdash s : A$ ?
- (ii) Given a term  $s$ , is there some type  $A$  such that  $\Gamma \vdash s : A$ ?

(These are decidable either if the context  $\Gamma$  is given, or if we ask the question of whether there is any such  $\Gamma$ .)

The question: “given a type  $A$ , is there some term  $s$  with  $\Gamma \vdash s : A$ ?” is trivial, because we can just use a variable of type  $A$ . But the question “given a type  $A$ , is there some *closed* term  $s$  with  $\vdash s : A$ ?” is decidable. This is not a consequence of the principal type algorithm, though – it is quite tricky to prove and the interested reader is directed to Chapter 8 (in particular 8D) of [Hin97].

Finally, we mention that there is a **converse principal type algorithm**, which constructs terms with a given *principal* type from some term which

can be given that type. Every type which is a type of some term is the principal type of some term, but in some cases (depending on the type) the term might have to be not in normal form. The interested reader is directed to 8B of [Hin97].

---

## Computational Practice

**8.a** Perform the following type substitutions:

- (i)  $(a \Rightarrow a)[(b \Rightarrow b)/a]$ .
- (ii)  $(a \Rightarrow a)[(b \Rightarrow a \Rightarrow b)/a]$ .
- (iii)  $((a \Rightarrow b) \Rightarrow b)[b/a, a/b]$ .
- (iv)  $((a \Rightarrow b) \Rightarrow b \Rightarrow a)[(c \Rightarrow b)/a, (a \Rightarrow c)/b, b/c]$ .

**8.b** Compute  $\mathbb{S} \circ \mathbb{T}$  where

- (i)  $\mathbb{S} = [a/b, b/a], \mathbb{T} = [c/b, c/a]$ .
- (ii)  $\mathbb{S} = [(a \Rightarrow c)/b, (b \Rightarrow c)/a, d/c], \mathbb{T} = [c/b, c/a]$ .

**8.c** Which of these pairs of types have unifiers? Where they do, compute a most general unification:

- (i)  $a \Rightarrow a$  and  $a \Rightarrow b \Rightarrow b \Rightarrow b$ .
- (ii)  $(a \Rightarrow b) \Rightarrow c$  and  $c \Rightarrow b \Rightarrow a$ .
- (iii)  $(a \Rightarrow a) \Rightarrow a$  and  $b \Rightarrow b \Rightarrow b$ .

**8.d** Compute the principal types of:

- (i)  $\ulcorner 0 \urcorner$ .
  - (ii)  $\ulcorner 1 \urcorner$ .
  - (iii)  $\ulcorner 2 \urcorner$ .
  - (iv)  $\lambda xy.(\lambda z.x)(yx)$ .
- 

## Exercises

**8.1** Let  $\mathbb{S}$  be a substitution and  $A$  a type. Show that there is a sequence of substitutions each with only a single component  $\mathbb{T}_i \equiv [B_i/b_i]$  for  $i = 1 \dots m$ , such that

$$\mathbb{S}(A) \equiv (\mathbb{T}_1 \circ \dots \circ \mathbb{T}_m)(A).$$

**8.2** Show that it is necessary for the above sequence of substitutions to depend on  $A$  as well as  $\mathbb{S}$  (so this is not a general result decomposing all substitutions into compositions of singleton substitutions).

**8.3** Show, by giving an algorithm, that the simple type **matching** problem is decidable: given types  $A$  and  $B$  find a substitution such that  $\mathbb{S}(A) \equiv B$ , or determine that no such substitution exists.

[Probably easiest is to modify Robinson's unification algorithm.]

To what extent is the substitution  $\mathbb{S}$  unique, if one exists?

**\*8.4** Recall the the type system with equality  $\text{TA}_{\lambda+\beta}$ , explored in Exercises 7.4-7.6. Prove that every term  $s$  typable in  $\text{TA}_{\lambda+\beta}$  has the same principal type that the  $\beta$ -normal form of  $s$  has in  $\text{TA}_{\lambda}$ .

**\*8.5** Show that  $a \Rightarrow n \Rightarrow d \Rightarrow r \Rightarrow e \Rightarrow w$  and  $k \Rightarrow e \Rightarrow r$  are not unifiable. What about  $l \Rightarrow a \Rightarrow m \Rightarrow b \Rightarrow d \Rightarrow a$  and  $c \Rightarrow a \Rightarrow l \Rightarrow c \Rightarrow u \Rightarrow l \Rightarrow u \Rightarrow s$ .

What is the general result here?



# Appendix A

## Answers to Computational Practice Questions

### Chapter 1

#### 1.a

- (i) A term.
- (ii) Not a term (two adjacent  $\lambda$ s).
- (iii) A term, more commonly written  $\lambda yx.x$ .
- (iv) A term, more commonly written  $\lambda xy.xy(\lambda z.z)$  or  $\lambda xy.xy\lambda z.z$
- (v) Not a term (unbalanced brackets).

#### 1.b

- (i)  $\lambda xyz.z(xy)$ .
- (ii)  $\lambda y.yyyy$ .
- (iii)  $\lambda y.yy((\lambda x.x)\lambda x.x)$ .

#### 1.c

- (i) One occurrence each of  $x$ ,  $y$ ,  $z$ , all bound.
- (ii) Two occurrences of  $y$ ,  $z$  and one of  $x$ . All bound.
- (iii) One bound occurrence of  $y$ . One free occurrence of  $z$  (the first) and two bound occurrences of  $z$ . This term violates the variable convention and we might rename the later  $z$ 's to something else.

#### 1.d

- (i)  $xyy$ .
- (ii)  $\lambda x.x$  (nothing to substitute for).

- (iii)  $\lambda y.zzy$ .
- (iv) Must rename the bound  $y$ 's first.  $\lambda u.yyu$ .
- (v)  $\lambda z.zz$ .
- (vi)  $\lambda xy.y(zy)y$  (note the brackets).

**1.e**

(i)

$$\frac{}{(\lambda x.x)(\lambda y.yy) = \lambda y.yy} (\beta)$$

(because  $x[(\lambda y.yy)/x] \equiv \lambda y.yy$ .)

(ii)

$$\frac{\frac{}{(\lambda x.x)(\lambda q.q) = \lambda q.q} (\beta)}{\lambda q.q = (\lambda x.x)(\lambda q.q)} (\text{sym})}{\lambda pq.q = \lambda p.(\lambda x.x)(\lambda q.q)} (\text{abs})$$

(iii)

$$\frac{\frac{\frac{}{(\lambda xy.y)p = \lambda y.y} (\beta)}{(\lambda xy.y)pq = (\lambda y.y)q} (\text{app})}{(\lambda xy.y)pq = q} (\text{trans})}{(\lambda xy.y)pq = q} (\text{trans})$$

(iv)

$$\frac{\frac{\text{(see above)}}{(\lambda xy.y)pq = q} (\text{abs})}{\lambda q.(\lambda xy.y)pq = \lambda q.q} (\text{abs})}{\lambda pq.(\lambda xy.y)pq = \lambda pq.q} (\text{abs})$$

Remember that  $\lambda pq.q$  and  $\lambda ab.b$  are the same term.  $\alpha$ -conversion is not part of the equational theory  $\lambda\beta$ , it's built into the definition of terms.

## Chapter 2

### 2.a

- (i) False; False; True.
- (ii) False; False; True.
- (iii) False; True; True.
- (iv) True; True; True.

### 2.b

- (i)  $\mathbf{k}\mathbf{k} \equiv (\lambda xy.x)(\lambda xy.x)$  is one redex. The whole term is another.
- (ii)  $\mathbf{\Omega} \equiv (\lambda x.xx)(\lambda x.xx)$  is a redex. So is  $(\lambda z.z)x$ . The whole term is another.
- (iii)  $(\lambda y.(\lambda z.(\lambda p.p)z)y)x$ ,  $(\lambda z.(\lambda p.p)z)y$ , and  $(\lambda p.p)z$ .
- (iv)  $\mathbf{y}$  has the redex  $(\lambda x.f(xx))(\lambda x.f(xx))$  as a subterm, and  $\mathbf{y}\mathbf{i}$  is another redex.

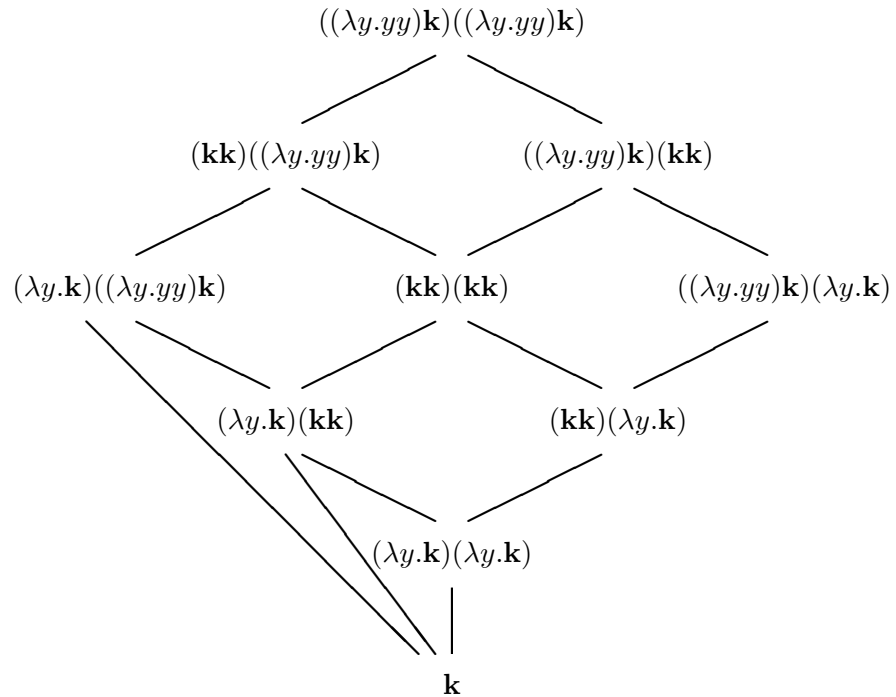
### 2.c

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

### 2.d

- (i)  $(\lambda xy.yy)\mathbf{\Omega}\mathbf{i}$  reduces to itself or to  $(\lambda y.yy)\mathbf{i}$ , which only reduces to  $\mathbf{ii}$ , which only reduces to  $\mathbf{i}$ , which is a normal form.

(ii)



(iii) This is a trick question. There are three redexes in the term so it looks like there are three possible one-step reducts. But in fact they are all  $\alpha$ -equivalent, i.e. the same term. The same happens at all future reduction steps. So there is a unique reduction path  $\lambda x.(\lambda y.(\lambda z.(\lambda p.p)z)y)x \rightarrow_{\beta} \lambda x.(\lambda z.(\lambda p.p)z)x \rightarrow_{\beta} \lambda x.(\lambda p.p)x \rightarrow_{\beta} \lambda x.x$ .

**2.e**

- (i) Yes  $(\lambda x.x)$ .
- (ii) Yes  $(\mathbf{k})$ .
- (iii) Yes  $(\lambda x.xx)$ .
- (iv) No.

## Chapter 3

### 3.a

(i)  $(\lambda x.xx)\mathbf{k}\Omega \xrightarrow{\beta} \mathbf{k}\mathbf{k}\Omega \xrightarrow{\beta} (\lambda y.\mathbf{k})\Omega \xrightarrow{\beta} \mathbf{k}$ . This is the  $\beta$ -normal form.

(ii)  $(\lambda xy.xxy)(\lambda xy.xxy)((\lambda xy.xyy)(\lambda xy.xyy))$   
 $\xrightarrow{\beta} (\lambda y.(\lambda xy.xxy)(\lambda xy.xxy)y)((\lambda xy.xyy)(\lambda xy.xyy))$   
 $\xrightarrow{\beta} (\lambda xy.xxy)(\lambda xy.xxy)((\lambda xy.xyy)(\lambda xy.xyy))$

A circular leftmost reduction path, so no  $\beta$ -normal form.

(iii)  $(\lambda x.xx)(\lambda y.\mathbf{k}(yyy)\mathbf{i})$   
 $\xrightarrow{\beta} (\lambda y.\mathbf{k}(yyy)\mathbf{i})A$  where  $A \equiv \lambda y.\mathbf{k}(yyy)\mathbf{i}$   
 $\xrightarrow{\beta} \mathbf{k}(AAA)\mathbf{i}$   
 $\xrightarrow{\beta^2} AAA$   
 $\equiv (\lambda y.\mathbf{k}(yyy)\mathbf{i})AA$   
 $\xrightarrow{\beta} \dots$

Now  $AA$  will always be a subterm, so the leftmost reductions continue forever. Therefore no  $\beta$ -normal form.

### 3.b

(i) No. Head reduces to itself, so no finite head reduction sequence.

(ii) Yes. In hnf.

(iii) No. Infinite head reduction sequence ( $\lambda x.xxx$ 's keep growing).

(iv) Yes. Head reduces to  $\lambda xy.x\Omega\Omega$  in one step, and this is a hnf.

### 3.c

(i) No. The leftmost  $\lambda$  is frozen after the first reduction.

(ii) Yes.

**3.d** The standard reduction sequence is

$$\begin{aligned} (\lambda x.xx)((\lambda y.yy)((\lambda z.zz)\mathbf{k})) &\rightarrow_{\beta} (\lambda y.yy)((\lambda z.zz)\mathbf{k})((\lambda y.yy)((\lambda z.zz)\mathbf{k})) \\ &\rightarrow_{\beta} (\lambda z.zz)\mathbf{k}((\lambda z.zz)\mathbf{k})((\lambda y.yy)((\lambda z.zz)\mathbf{k})) \\ &\rightarrow_{\beta} \mathbf{k}\mathbf{k}((\lambda z.zz)\mathbf{k})((\lambda y.yy)((\lambda z.zz)\mathbf{k})) \\ &\rightarrow_{\beta} (\lambda p.\mathbf{k})((\lambda z.zz)\mathbf{k})((\lambda y.yy)((\lambda z.zz)\mathbf{k})) \\ &\rightarrow_{\beta} \mathbf{k}((\lambda y.yy)((\lambda z.zz)\mathbf{k})) \\ &\rightarrow_{\beta} \mathbf{k}((\lambda z.zz)\mathbf{k})((\lambda z.zz)\mathbf{k}) \\ &\rightarrow_{\beta} \mathbf{k}(\mathbf{k}\mathbf{k}((\lambda z.zz)\mathbf{k})) \\ &\rightarrow_{\beta} \mathbf{k}((\lambda q.\mathbf{k})((\lambda z.zz)\mathbf{k})) \\ &\rightarrow_{\beta} \mathbf{k}\mathbf{k} \end{aligned}$$

But the shortest reduction sequence is

$$\begin{aligned}
 (\lambda x.xx)((\lambda y.yy)((\lambda z.zz)\mathbf{k})) &\rightarrow_{\beta} (\lambda x.xx)((\lambda y.yy)(\mathbf{k}\mathbf{k})) \\
 &\rightarrow_{\beta} (\lambda x.xx)((\lambda y.yy)(\lambda p.\mathbf{k})) \\
 &\rightarrow_{\beta} (\lambda x.xx)((\lambda p.\mathbf{k})(\lambda p.\mathbf{k})) \\
 &\rightarrow_{\beta} (\lambda x.xx)\mathbf{k} \\
 &\rightarrow_{\beta} \mathbf{k}\mathbf{k}
 \end{aligned}$$

(In this case the shortest reduction strategy is to reduce the rightmost redex at each step.)

### 3.e

- (i) Already in hnf. Use  $t_1 \equiv t_2 \equiv \lambda z_1 z_2 x.x$ .
  - (ii) The hnf is  $\lambda y.y(\lambda xy.yxxy)(\lambda xy.yxxy)y$ . Use  $t_1 \equiv \lambda z_1 z_2 z_3 x.x$
  - (iii) The hnf is  $\lambda y.y\mathbf{\Omega}\mathbf{\Omega}$ . Use  $t_1 \equiv \lambda z_1 z_2 x.x$ .
- (There are many other correct answers of course.)

**Chapter 4**

**4.a**  $\lceil 4 \rceil$ .

**4.b**  $f(x) = \begin{cases} 0, & \text{if } x = 0 \\ 1, & \text{if } x > 0 \end{cases}$

**4.c**  $f(x) = 2x$ .

**4.d**  $\text{not} \equiv \lambda bxy.byx$ .

**Chapter 5**

**5.a**

- (i)  $\mathbf{S(Ky)(SKK)}$ .
- (ii)  $\mathbf{S(KK)(SKK)}$ .
- (iii)  $\mathbf{S(K(S(SKK)))(S(KK)(SKK))}$ .

**5.b**

- (i)  $\mathbf{S(K(S(SKK)))(S(KK)(SKK))}$ .
- (ii)  $\mathbf{S(SKK)(K(SKK))}$ .
- (iii)  $\mathbf{SKKy}$ .

**5.c**  $ss; \lambda abc.bc(abc); \Omega$ .

**5.d**

- (i)  $\mathbf{KS}$ .
- (ii)  $\mathbf{SS(SS)}$ .
- (iii)  $\mathbf{SSS}$ .

## Chapter 6

### 6.a

- (i)  $(A \Rightarrow (A \Rightarrow (A \Rightarrow A)))$ .
- (ii)  $(A \Rightarrow ((B \Rightarrow C) \Rightarrow A))$ .
- (iii)  $(A \Rightarrow ((B \Rightarrow C) \Rightarrow ((A \Rightarrow D) \Rightarrow E)))$ .

### 6.b

(i)

$$\frac{\frac{\overline{\{y:A\} \mapsto y : A} \quad \overline{\{x:A \Rightarrow B\} \mapsto x : A \Rightarrow B}}{\overline{\{y:A, x:A \Rightarrow B\} \mapsto xy : B}}}{\overline{\{y:A\} \mapsto \lambda x.xy : (A \Rightarrow B) \Rightarrow B}}$$

(ii)

$$\frac{\frac{\overline{\{x:A \Rightarrow A\} \mapsto x : A \Rightarrow A} \quad \overline{\{y:A\} \mapsto y : A}}{\overline{\{x:A \Rightarrow A, y:A\} \mapsto xy : A}}}{\overline{\{x:A \Rightarrow A\} \mapsto \lambda y.xy : A \Rightarrow A}}$$

(iii)

$$\frac{\frac{\overline{\{x:B \Rightarrow C\} \mapsto x : B \Rightarrow C} \quad \overline{\{y:B\} \mapsto y : B}}{\overline{\{x:B \Rightarrow C, y:B\} \mapsto xy : C}}}{\overline{\{x:B \Rightarrow C\} \mapsto \lambda y.xy : B \Rightarrow C}}$$

(iv)

$$\frac{\frac{\overline{\{x:B \Rightarrow C\} \mapsto x : B \Rightarrow C} \quad \frac{\overline{\{y:A \Rightarrow B\} \mapsto y : A \Rightarrow B} \quad \overline{\{z:A\} \mapsto z : A}}{\overline{\{y:A \Rightarrow B, z:A\} \mapsto yz : B}}}{\overline{\{x:B \Rightarrow C, y:A \Rightarrow B, z:A\} \mapsto x(yz) : C}}}{\overline{\{y:A \Rightarrow B, z:A\} \mapsto \lambda x.x(yz) : (B \Rightarrow C) \Rightarrow B}}}{\overline{\{z:A\} \mapsto \lambda x.x(yz) : (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow B}}}{\overline{\mapsto \lambda zyx.x(yz) : A \Rightarrow (A \Rightarrow B) \Rightarrow (B \Rightarrow C) \Rightarrow C}}$$

### 6.c

- (i)  $A \Rightarrow B \Rightarrow C \Rightarrow B$ .
- (ii)  $A \Rightarrow (A \Rightarrow B) \Rightarrow B$ .
- (iii)  $(A \Rightarrow B \Rightarrow C) \Rightarrow A \Rightarrow B \Rightarrow C$ .
- (iv)  $((A \Rightarrow A) \Rightarrow B) \Rightarrow B$ .



**Chapter 8****8.a**

- (i)  $(b \Rightarrow b) \Rightarrow b \Rightarrow b$ .
- (ii)  $(b \Rightarrow a \Rightarrow b) \Rightarrow b \Rightarrow a \Rightarrow b$ .
- (iii)  $(b \Rightarrow a) \Rightarrow a$ .
- (iv)  $((c \Rightarrow b) \Rightarrow a \Rightarrow c) \Rightarrow (a \Rightarrow c) \Rightarrow c \Rightarrow b$

**8.b**

- (i)  $[c/b, c/a]$ .
- (ii)  $[d/a, d/b, d/c]$ .

**8.c**

- (i) Unifiable.  $(b \Rightarrow b \Rightarrow b) \Rightarrow b \Rightarrow b \Rightarrow b$ .
- (ii) Unifiable.  $(b \Rightarrow b) \Rightarrow b \Rightarrow b$ .
- (iii) Not unifiable.

**8.d**

- (i)  $a \Rightarrow b \Rightarrow a$ .
- (ii)  $(a \Rightarrow b) \Rightarrow a \Rightarrow b$ .
- (iii)  $(a \Rightarrow a) \Rightarrow a \Rightarrow a$ .
- (iv)  $a \Rightarrow (a \Rightarrow b) \Rightarrow a$ .



## Appendix B

# Sample Finals Questions

Here are some undergraduate finals-style practice questions. We will use them, in Trinity Term, for a revision class. It would be best to wait until then, before attempting them.

### Question 1

- (a) Define the *diamond property* and the *Church-Rosser property* of a notion of reduction  $\rightarrow_R$ . (2 marks)
- (b) Does  $\beta$ -reduction have the diamond property? Give a proof or counterexample. (4 marks)
- (c) Assuming that the Church-Rosser property holds for  $\beta$ -reduction, show that no term can have more than one  $\beta$ -normal form. (2 marks)

Define the *reduction graph*  $G(s)$  of a term  $s$  to be the directed graph  $(N, E)$ , where the set of nodes  $N = \{t \mid s \rightarrow_\beta t\}$  and edges  $E = \{(t, u) \mid t \rightarrow_\beta u\}$ .

- (d) Draw the reduction graph of the term  $(\lambda x.xx)(\lambda y.(\lambda x.xx)y)$ . (3 marks)
- (e) Give an example of a term which has no  $\beta$ -normal form, but whose reduction graph is acyclic. (4 marks)
- (f) Which of the following equations hold in  $\lambda\beta$ ? Justify your answers. If you wish you may assume that two terms are equal if and only if they have a common  $\beta$ -reduct.

i)  $(\lambda x.x)(\lambda xy.yy) = (\lambda x.xx)(\lambda xy.yy)$  (3 marks)

ii)  $(\lambda x.xx)(\lambda xy.yy) = (\lambda x.xxx)(\lambda xy.yy)$  (3 marks)

iii)  $AA = AA(\lambda p.p)$ , where  $A \equiv \lambda xy.y(xx)$  (4 marks)

iv)  $AA = AA(\lambda pq.qp)$ , where  $A \equiv \lambda xy.y(xx)$  (4 marks)

[adapted from 2004 II.1 Question 7 and 2005 Section C]

## Question 2

We make the following definition for the representation of (finite) tuples in the untyped  $\lambda$ -calculus: for each  $n \geq 1$ ,

$$\langle s_1, \dots, s_n \rangle \equiv \lambda x.xs_1 \dots s_n$$

(where  $x$  is a fresh variable not occurring in any of the terms  $s_i$ ).

- (a) Show that  $\lambda\beta \vdash \langle s_1, \dots, s_n \rangle = \langle t_1, \dots, t_n \rangle$  if and only if for each  $1 \leq i \leq n$ ,  $\lambda\beta \vdash s_i = t_i$ . (2 marks)
- (b) What is a *fixed point combinator*? Give two examples. (3 marks)
- (c) Find a closed term  $s$  satisfying  $\lambda\beta \vdash \langle s, s \rangle = s$ . (3 marks)
- (d) Find a closed term  $t$  such that  $\lambda\beta \vdash \{ \langle t, t \rangle = t \}$  is inconsistent. (6 marks)

[You should not use Böhm's Theorem. Try to find a term satisfying  $\lambda\beta \vdash tu = t$ , for all  $u$ .]

- (e) For fixed  $n$ , describe terms  $\tau$  and  $\pi_1, \dots, \pi_n$  which satisfy  $\lambda\beta \vdash \tau s_1 \dots s_n = \langle s_1, \dots, s_n \rangle$ , and  $\lambda\beta \vdash \pi_i \langle s_1, \dots, s_n \rangle = s_i$  for each  $i$  and any terms  $s_1, \dots, s_n$ .  
Hence prove the following *n-ary fixed point theorem*: for any terms  $p_1, \dots, p_n$  there exist terms  $f_1, \dots, f_n$  satisfying  $\lambda\beta \vdash p_i f_1 f_2 \dots f_n = f_i$  for each  $i$ . (7 marks)
- (f) Hence, or otherwise, find terms  $f$  and  $g$  satisfying  $\lambda\beta \vdash f = \langle f, g \rangle$  and  $\lambda\beta \vdash g = \langle g, f \rangle$ . (4 marks)

[2003 II.5 Question 5]

### Question 3

- (a) Define the equational theory  $\lambda\beta$ .
- (b) Recall that the theory  $\lambda\beta\eta$  is obtained by adding the  $\eta$  rule

$$\frac{}{\lambda x.sx = s} \quad (x \text{ not free in } s)$$

to the system  $\lambda\beta$ .

Prove that  $\lambda\beta \not\vdash y = \lambda x.yx$  [You may assume standard results connecting equality in  $\lambda\beta$  with common  $\beta$ -reducts.] but that this equation does hold in  $\lambda\beta\eta$ .

- (c) Consider the terms of combinatory logic, with the usual constants **K** and **S**.

We define an equational theory on the terms, which is strengthened by an *extensionality rule*:

$$\frac{}{A = A} \quad \frac{A = B}{B = A} \quad \frac{A = B \quad B = C}{A = C}$$

$$\frac{\mathbf{K}AB = A \quad \mathbf{S}ABC = AC(BC)}{A = A' \quad B = B'} \quad \frac{}{AB = A'B'}$$

$$\frac{Ax = Bx}{A = B} \quad (x \text{ does not occur in } A \text{ or } B)$$

We write  $ECL \vdash A = B$  if  $A = B$  is derivable in this system.

Prove that  $ECL \vdash \mathbf{SK} = \mathbf{K}(\mathbf{SKK})$ .

- (d) The usual translations  $(-)_\lambda$  and  $(-)_{cl}$  between the terms of the  $\lambda$ -calculus and the terms of combinatory logic are given by the following:

$$\begin{array}{ll} x_{cl} \equiv x & x_\lambda \equiv x \\ (st)_{cl} \equiv s_{cl}t_{cl} & (AB)_\lambda \equiv A_\lambda B_\lambda \\ (\lambda x.s)_{cl} \equiv \lambda x.(s_{cl}) & \mathbf{K}_\lambda \equiv \lambda xy.x \\ & \mathbf{S}_\lambda \equiv \lambda xyz.xz(yz) \end{array}$$

(where  $\lambda x.A$  denotes an abstraction algorithm; you may assume that  $ECL \vdash (\lambda x.A)B = A[B/x]$ .)

Prove the following results, which connect  $\lambda\beta\eta$  with  $ECL$ :

- i)  $\lambda\beta\eta \vdash (s_{cl})_\lambda = s$ , for all terms  $s$ .  
 [You may assume that  $\lambda\beta \vdash (\lambda x.A)_\lambda = \lambda x.A_\lambda$ .]

- ii)  $ECL \vdash (A_\lambda)_{cl} = A$ , for all CL-terms  $A$ .
- iii)  $ECL \vdash A = B$  implies  $\lambda\beta\eta \vdash A_\lambda = B_\lambda$  for all CL-terms  $A$  and  $B$ .

(e) Can (a) be strengthened to  $\lambda\beta \vdash (s_{cl})_\lambda = s$ ?

(f) Can (c) be strengthened to  $ECL \vdash A = B$  implies  $\lambda\beta \vdash A_\lambda = B_\lambda$ ?

[2005 section C]

#### Question 4

Let  $\sharp$  be an effective Gödel numbering of the terms of the untyped  $\lambda$ -calculus, and let  $\ulcorner n \urcorner$  be the  $n$ -th Church numeral. For any term  $s$  we write  $\ulcorner s \urcorner$  for  $\ulcorner \sharp s \urcorner$ .

- (a) What does it mean to say that a set  $S \subseteq \Lambda$  is *recursive*?

State the Second Recursion Theorem for the untyped  $\lambda$ -calculus.

(4 marks)

- (b) Let  $S$  be a nonempty set of terms, not equal to the whole of  $\Lambda$ , which is closed under equality. Show that  $S$  is not recursive. (10 marks)
- (c) For any term  $s$  define the set

$$S_s = \{t \in \Lambda \mid st = s\},$$

and set  $\mathbf{k} \equiv \lambda xy.x$ .

- i) Let  $u \equiv \lambda x.\mathbf{k}$ . Show that  $S_u$  is empty.
- ii) Let  $\mathbf{y}$  be any fixed point combinator. Show that  $S_{\mathbf{y}\mathbf{k}} = \Lambda$ .
- iii) Let  $\mathbf{i} \equiv \lambda x.x$ . Show that  $S_{\mathbf{i}}$  is not recursive.

(4+2+5 marks)

[You may use standard results about the definability of partial functions in the untyped  $\lambda$ -calculus, and the consistency of the theory  $\lambda\beta$ , without proof. If you use the Scott-Curry theorem you must prove it.]

[2001 II.5 Question 6]

**Question 5**

- (a) What is a *principal type* of a term  $s$ ? Is it unique? (3 marks)
- (b) Describe the inputs and outputs of a *Principal Type Algorithm*. (2 marks)
- (c) State the *Strong Normalization Theorem* for the simple type system  $TA_\lambda$ . (2 marks)
- (d) Which of the following are decidable properties of the term  $s$ ? Justify your answers. You may use any standard results about decidability in the untyped  $\lambda$ -calculus, and standard properties of the type system  $TA_\lambda$ , as long as you state them clearly first. You may also assume Church's Thesis.
- i)  $s$  is typable. (2 marks)
  - ii)  $s$  is typable to a specific type  $A$ . (3 marks)
  - iii)  $s$  is equal (under  $\lambda\beta$ ) to some typable term. (6 marks)
  - iv)  $s$  reduces (under  $\beta$ -reduction) to some typable term. (7 marks)

**Question 6**

This question refers to the simple type theory  $TA_\lambda$ .

- (a) Give a type deduction showing that

$$\mapsto \mathbf{ki} : (B \Rightarrow B) \Rightarrow A \Rightarrow A,$$

for any types  $A$  and  $B$  (where  $\mathbf{k} \equiv \lambda xy.x$  and  $\mathbf{i} \equiv \lambda x.x$ ). (4 marks)

- (b) What is the principal type of  $\mathbf{ki}$ ? (2 marks)

- (c) Let  $s$  be some **closed** term. For each of the following statements give a proof or counterexample.

i) If  $s$  is typable then  $\lambda x.s$  is typable. (3 marks)

ii) If  $s$  is typable then  $\mathbf{is}$  is typable. (4 marks)

iii) If  $s$  is typable then  $\mathbf{si}$  is typable. (4 marks)

iv) If  $s$  is solvable then  $s$  is typable.

[Recall that a closed term  $s$  is **solvable** if there exist terms  $t_1, \dots, t_n$  satisfying  $\lambda\beta \vdash st_1 \dots t_n = \mathbf{i}$ .] (4 marks)

v) If  $s$  is typable then  $s$  is solvable. (4 marks)

*You may assume any standard results about the type system  $TA_\lambda$ , as long as you state them clearly. If you claim that a certain term is untypable then you must prove it.*

[2005 B2 Question 3]



## Other Past Questions Suitable for Practice

Lambda calculus has been a course for some years, but until 2003-4 the syllabus was different. There were separate Mathematics and CS papers (section C and paper B2, respectively) in 2005, both of which are still suitable for practice, and three questions set in 2006–2008.

Some even older finals questions (all taken from Computer Science papers) are still suitable:

2000 II.5 Questions 1, 2

2001 II.5 Question 5 (question 7 is also worth practicing, but is too focussed on the topic of solvability, which only just gets in to this year's syllabus)

2002 II.5 Questions 9, 10, 12

(Older questions on combinatory logic are reasonable but the terminology is different to that which I now use. Older types questions were perfectly nice, but it was a different type system.)



## Appendix C

# A Primer On Computability

This goes over some basic definitions and background on computability, for Chapter 4. It does not present the material in a historically accurate order, and introduces many concepts in a rough-and-ready way. I encourage you to learn about computability properly, if you haven't before.

First, a word about functions. We will concentrate on the so-called **numeric functions**, which are just functions  $\phi : \mathbb{N}^m \rightarrow \mathbb{N}$  for some  $m$ . Since there is an easy bijection between  $\mathbb{N}^m$  and  $\mathbb{N}$ , we can actually restrict our attention to functions on the natural numbers. We are also interested in the **partial** numeric functions (functions which might be undefined on some parts of the domain). When we want to emphasise that a function is defined on the whole domain, we say that it is **total**.

Here is some standard notation for partial functions:

1. We write  $\phi(n_1, \dots, n_m) = k$  to mean that  $\phi$  is defined on  $\langle n_1, \dots, n_m \rangle$  and equal to  $k$ .
2. We write  $\phi(n_1, \dots, n_m) \downarrow$  to mean that  $\phi$  is defined on  $\langle n_1, \dots, n_m \rangle$ .
3. We write  $\phi(n_1, \dots, n_m) \uparrow$  to mean that  $\phi$  is not defined on  $\langle n_1, \dots, n_m \rangle$ .
4. For two functions  $\phi$  and  $\chi$  we write  $\phi(n_1, \dots, n_m) \simeq \chi(n_1, \dots, n_m)$  to mean that one is defined on  $\langle n_1, \dots, n_m \rangle$  if and only if the other is, and if they are defined they are equal.

It is an initially surprising fact that there are some (partial) functions which can never be computed by any machine, regardless of memory or time limitations. Our starting point is the Turing Machine.

**Definition** A **Turing Machine** is an imaginary device which consists of a readable and writable tape divided up into individual units, a read/write

head, and a finite set of states including a unique initial state and a unique halting state.

The tape is unbounded in length. Each unit of the tape can be written with an encoding of the symbols 0,1, or blank. Reading and writing of the units is accomplished by the head.

The machine begins computing in the following way — the head is positioned at some identifiable origin on the tape, and the tape may have data written on it already. The machine is in the initial state. At each time step (time is considered to be discrete) the machine reads the symbol written on the unit of the tape under the head.

The Turing Machine's **program** is map which determines, given the current state and the symbol read by the head, whether a) to write a new symbol to the unit of the tape currently under the head, then b) whether to move the head one unit to the left or right and then c) which state to change to.

The machine continues this process until it reaches the halting state, or forever if the halting state is never reached.

The definition is obviously motivated by the sort of computer in use when Turing was considering this notion! But it serves as a good example of a completely “basic” machine. The machine computes a partial function (on the natural numbers) as follows: suppose that initially the tape consists of blanks except for a string of  $n$  repetitions of the symbol 1, starting at the tape origin, and that when started from this state the machine eventually halts with the tape head at the start of a string of  $m$  consecutive 1 symbols. We say that the machine has input the number  $n$  and output the number  $m$ . If for every natural number  $n$  the machine outputs  $f(n)$  then that program **computes** the function  $f$ .

A few remarks: the program need not guarantee that the machine ever halts. It is quite possible for the program to get stuck in a loop. However the number of states, and hence the size of the program (which is just a look-up table saying from each state and read under the current head position what to do next) is finite. For a machine to compute a total function it must always eventually halt, not matter what number is input.

We can extend this to cope with partial functions — we say that a machine **(strongly) computes** a partial function  $f$  if it computes  $f$  on all points of the domain at which  $f$  is defined, and fails to halt when input a number at which  $f$  is undefined.

It is routine to show that there are only countably many Turing Machines (because the number of states must be finite). So by enumerating them we can talk about “the  $n^{\text{th}}$  Turing Machine”.

Now there are some functions which cannot be computed by any Turing Machine. A classic example is the **Halting Problem**, which is the function

$\phi : \mathbb{N} \rightarrow \mathbb{N}$  given by

$$\phi(n) = \begin{cases} 1, & \text{if the } n^{\text{th}} \text{ Turing Machine eventually halts when given the input } n \\ 0, & \text{if the } n^{\text{th}} \text{ Turing Machine does not ever halt when given the input } n. \end{cases}$$

(Here is a quick sketch of why it is not computable by any Turing Machine. Suppose it was computable by a Turing Machine  $T$ ; take  $T$  and make one modification – when it was about to enter the halting state after outputting 1, instead make it enter a loop. Call the modified machine  $T'$ ;  $T'$  has the property that it terminates on input  $m$  if and only if the  $m^{\text{th}}$  Turing Machine does not terminate on input  $m$ . Suppose that  $T'$  is the  $k^{\text{th}}$  Turing Machine; feed it input  $k$ . We have a machine which terminates if and only if it does not terminate.)

The first theorem of computability is a characterisation of those partial (and total) functions which can be computed by Turing machines.

**Definition** Define the set of functions  $\mathcal{R} \subseteq \{\phi \mid \phi : \mathbb{N}^m \rightarrow \mathbb{N} \text{ for some } m \in \mathbb{N}\}$  recursively as follows:

- (Initial functions) The functions  $\sigma : n \mapsto n + 1$ ,  $\zeta : n \mapsto 0$  and for each  $1 \leq j \leq i$ ,  $\Pi_j^i : \langle n_1, \dots, n_i \rangle \mapsto n_j$  are in  $\mathcal{R}$ .
- (Closure under Composition) If  $\chi : \mathbb{N}^m \rightarrow \mathbb{N} \in \mathcal{R}$  and for each  $1 \leq i \leq m$ ,  $\psi_i : \mathbb{N}^l \rightarrow \mathbb{N} \in \mathcal{R}$ , then  $\phi : \mathbb{N}^l \rightarrow \mathbb{N} \in \mathcal{R}$ , where  $\phi(n_1, \dots, n_l) \downarrow$  if and only if all  $\psi_i(n_1, \dots, n_l) \downarrow$  and  $\chi(\psi_1(\vec{n}), \dots, \psi_m(\vec{n})) \downarrow$ , in which case

$$\phi(n_1, \dots, n_l) = \chi(\psi_1(\vec{n}), \dots, \psi_m(\vec{n})).$$

- (Closure under Primitive Recursion) If  $\chi : \mathbb{N}^m \rightarrow \mathbb{N} \in \mathcal{R}$ ,  $\psi : \mathbb{N}^{m+2} \rightarrow \mathbb{N} \in \mathcal{R}$  then  $\phi : \mathbb{N}^{m+1} \rightarrow \mathbb{N} \in \mathcal{R}$ , where  $\phi(0, n_1, \dots, n_m) \simeq \chi(n_1, \dots, n_m)$ , and  $\phi(k+1, n_1, \dots, n_m) \downarrow$  if and only if  $\phi(k, n_1, \dots, n_m) \downarrow$  and  $\chi(\phi(k, n_1, \dots, n_m), k, n_1, \dots, n_m) \downarrow$ , in which case

$$\phi(k+1, n_1, \dots, n_m) = \chi(\phi(k, n_1, \dots, n_m), k, n_1, \dots, n_m).$$

- (Closure under Minimalization) If  $\chi : \mathbb{N}^{m+1} \rightarrow \mathbb{N} \in \mathcal{R}$  then  $\phi : \mathbb{N}^m \rightarrow \mathbb{N} \in \mathcal{R}$ , where  $\phi(n_1, \dots, n_m) = k$  if there is some  $k \in \mathbb{N}$  such that for all  $0 \leq j < k$ ,  $\chi(j, n_1, \dots, n_m) \downarrow$  and  $\chi(j, n_1, \dots, n_m) \neq 0$  and  $\chi(k, n_1, \dots, n_m) = 0$ .  $\phi(n_1, \dots, n_m) \uparrow$  if no such  $k$  exists.

The functions  $\mathcal{R}$  are called the **partial recursive functions**. The subset  $\mathcal{R}_0$  of total functions in  $\mathcal{R}$  is called the **total recursive functions** and they can be given an inductive definition of their own (which can be found in Chapter 4).

The following theorem can then be proved:

**Theorem** A function is partial recursive if and only if it can be computed by some Turing Machine. A total function is recursive if and only if it can be computed by some Turing Machine.

To see some more examples of functions not in  $\mathcal{R}$ , and how to prove the above theorem, find an introductory book on computability and read it. An example of a theorem about Turing Machines is the following:

**Theorem** There is a **universal Turing Machine**. This is a machine which, when given the input pair  $\langle n, m \rangle$  outputs  $r$  if and only if the  $n^{\text{th}}$  Turing Machine outputs  $r$  on input  $m$  (where pairs are encoded into the natural numbers in some standard way).

Okay, so there is a limit to which functions can be computed by Turing Machines — perhaps the definition of Turing Machines was a bit restrictive and if we give the machines a little more power (maybe the ability to write symbols other than just 0 and 1, or multiple tapes) then they will be able to compute more functions? Actually this is not the case.

Since Turing Machines were described, many other abstract notions of computation have been proposed. If we use them to compute numeric functions, all of them have been shown to compute a set of partial functions contained in  $\mathcal{R}$ , usually  $\mathcal{R}$  itself.

Of course, it is easy to give an example of a computing device which cannot compute all functions in  $\mathcal{R}$  — a trivial machine which never does anything, for example — but actually as long as the machine can deal with partiality and a modest effort has been made to give it a reasonable amount of computing power (including that there is no limitation on memory or similar resources), it usually turns out that precisely the whole of  $\mathcal{R}$  can be computed. This body of evidence leads to the following belief:

**Thesis (Church's Thesis)** The **effectively computable partial functions**, those which can be computed by some device or algorithm, are the partial recursive functions.

This statement could never be proved although it could be disproved by finding a procedure to compute some function not in  $\mathcal{R}$ . This has not happened to date (and Church's Thesis was first mentioned in the 1930's). On the whole, computer scientists accept Church's Thesis as true. Thus to "prove" that a function is partial recursive we will just describe a procedure for computing it and appeal to Church's Thesis.

So here are the standard definitions associated with computability:

**Definition** A (partial) numeric function is **computable** if it is computable by a Turing Machine, **(partial) recursive** if it is in  $\mathcal{R}$ , and **effective** or **effectively computable** if it is computable by some device or algorithm. In view of the theorem above and Church's Thesis, these are all the same and the terminology is used interchangeably.

A predicate on the natural numbers  $P(n)$  is **decidable** if there is a total computable function  $\phi : \mathbb{N} \rightarrow \mathbb{N}$  satisfying:

$$\phi(n) = \begin{cases} 1, & \text{if } P(n) \text{ is true,} \\ 0, & \text{if } P(n) \text{ is false.} \end{cases}$$

A predicate on the natural numbers  $P(n)$  is **semi-decidable** if there is a partial computable function  $\phi : \mathbb{N} \rightarrow \mathbb{N}$  satisfying:

$$\phi(n) \begin{cases} = 1, & \text{if } P(n) \text{ is true,} \\ \text{undefined,} & \text{if } P(n) \text{ is false.} \end{cases}$$

A set  $S \subset \mathbb{N}$  is **decidable** or **recursive** if the membership predicate is decidable. i.e. there is a total computable function  $\phi : \mathbb{N} \rightarrow \mathbb{N}$  satisfying:

$$\phi(n) = \begin{cases} 1, & \text{if } n \in S, \\ 0, & \text{if } n \notin S. \end{cases}$$

A set  $S \subset \mathbb{N}$  is **semi-decidable** or **recursively enumerable** (often written r.e.) if the membership predicate is semi-decidable. i.e. there is a partial computable function  $\phi : \mathbb{N} \rightarrow \mathbb{N}$  satisfying:

$$\phi(n) \begin{cases} = 1, & \text{if } n \in S, \\ \text{undefined,} & \text{if } n \notin S. \end{cases}$$

We can extend these definitions to functions and predicates on a domain  $D$  other than  $\mathbb{N}$ , and subsets of a set  $D$  other than  $\mathbb{N}$ , as long as  $D$  is countable and hence can be encoded into  $\mathbb{N}$ . (Technical note: computability is really about  $\mathbb{N}$ , and if the bijection between  $D$  and  $\mathbb{N}$  is not "computable" in some sense then this might lead to unusual results.)

For examples of undecidable and semi-decidable predicates, and nonrecursive or r.e. sets, find a book about computability and read it.

In view of Church's Thesis, we could say the a predicate is decidable if there is a device or algorithm which computes its truth value from an input. A predicate is semi-decidable if there is a device or algorithm which guarantees to tell us if the predicate is true on a given input, but the device or algorithm does not terminate if the predicate is false.

[A good example of a semi-decidable predicate is whether, for given terms  $s$  and  $t$  of the untyped  $\lambda$ -calculus,  $\lambda\beta \vdash s = t$ . To see why it is at least

semi-decidable, use the following algorithm: starting bottom-up, produce in breadth-first order all possible proof trees terminating in  $s = t$ . We know that the proof is valid if we find a tree with no unproven hypotheses, but if there is no such tree then the algorithm will search forever without success, never terminating. This does not show that the predicate is not decidable, perhaps by a more clever algorithm, but see Exercise 4.10.]

So given a framework of computation it is natural to ask what (partial) functions can be computed in it. By Church's Thesis we would be very surprised if it computed anything not in  $\mathcal{R}$ . If it does not compute all functions in  $\mathcal{R}$  we might be disappointed, because the structure of the language has stopped us from computing some functions which other languages will allow. (If it only computes total functions then we would have to be satisfied with computing all functions in  $\mathcal{R}_0$ .) If a language can compute every function in  $\mathcal{R}$  then we say that it is **Turing Complete**. This is the case for the untyped  $\lambda$ -calculus, and it is not the case for the pure simply-typed language.



# Index

- $\Lambda$ , 2
- $\Lambda^0$ , 6
- $\alpha$ -convertible, 5
- $\beta$ -conversion, 17
- $\beta$ -convertibility, 6
- $\beta$ -reduction, 17
- $\beta$ -simulation, 55
- $\beta\delta_C$ -normal forms, 29
- $\delta_C$ , 29
- $\eta$ -conversion, 9
- $\eta$ -expansion, 18
- $\eta$ -reduction, 18
- $\lambda$ -algebra, 60
- $\lambda$ -definable, 44
- $\lambda$ -theory, 10
- $\lambda I$ -term, 18
- $\lambda\beta$ , 9
- $\lambda\beta\eta$ , 9
- $\mathcal{T}_{NF}$ , 24
- app**, 47
- b**, 6
- c**, 6
- f**, 6, 25
- gnum**, 47
- i**, 6
- k**, 6
- $\Omega$ , 6, 17
- rcase**, 44
- s**, 6
- succ**, 44
- t**, 6, 25
- zero?**, 44
  
- abstraction, 3
- abstraction algorithm, 55
- advancement of head reduction, 33
  
- affine, 18
- application, 3
- arrow types, 72
- atomic, 72
- axiom, 1
  
- Böhm's theorem, 25
- basis, 68
- bound, 5
  
- cancelling, 18
- characteristic function, 51
- Church numerals, 43
- Church-Rosser, 20
- CL-terms, 54
- closed, 5
- closure under composition, 45, 129
- closure under minimalization, 45, 129
- closure under primitive recursion, 45, 129
- combinatory algebra, 59
- combinatory complete, 61
- common instance, 97
- commute, 29
- compatible, 16
- compatible closure, 16
- component, 94
- composite types, 72
- composition, 95
- composition-extension, 96
- computable, 131
- computes, 128
- congruence, 9
- consistent, 10, 72
- construction tree, 4
- context, 7

- contextual substitution, 8
- continuous, 63
- contracted, 17
- converse principal type algorithm, 106
- conversion, 5
- convertibility, 5
  
- de Bruijn notation, 51, 88
- decidable, 131
- definability theorem, 45
- denotation, 59, 60
- diamond property, 20
- dummy variable, 5
- duplicating, 18
  
- easy, 41
- effective, 131
- effectively computable, 131
- effectively computable partial functions, 130
- empty substitution, 94
- environment, 59
- extended polynomials, 89
- extensional, 10
- extensional equality, 95
- extensionality rule, 57
  
- fixed point, 11
- fixed point combinator, 11
- free, 5
- free variables, 5
- fresh, 5
- function types, 72
  
- Gödel numbering, 47
- generated, 68
- ground types, 72
  
- Halting Problem, 128
- has a  $R$ -normal form, 19
- head normal form, 32
- head redex, 33
- head reduction, 32
- Hilbert-Post complete, 11, 42
- hnf, 32
  
- hole variables, 8
  
- implements, 18
- in  $R$ -normal form, 19
- in normal form, 19
- initial functions, 45, 129
- instance, 93
- internal reduction, 33
  
- Kleene's first model, 69
  
- leftmost reduction, 32
- linear, 18
  
- m.g.u., 98
- matching, 98
- maximally consistent, 11
- most general unification, 98
- most general unifier, 98
  
- normalising, 35
- normalizable (w.r.t.  $\rightarrow_R$ ), 19
- notion of reduction, 15
- numeral system, 51
- numeric functions, 127
  
- occurrence, 5
- one-holed, 17
- one-step  $R$ -reduction, 16
  
- parallel  $\beta$ -reduction, 21
- partial, 127
- partial functions, 48
- partial recursive functions, 129
- principal deduction, 93
- principal type, 93
- principal type algorithm, 100
- program, 128
  
- recursive, 131
- recursively enumerable, 131
- recursively inseparable, 51
- redex, 17
- redex rule, 15
- reducible, 85
- reduct, 16

- reduction sequence, 33
- reduction strategy, 20, 31
- reducts, 17
- Robinson's Unification Algorithm, 100
- Scott-Curry theorem, 47
- second recursion theorem, 47
- semi-decidable, 131
- semi-sensible, 39
- sensible, 39
- side condition, 1
- solvable, 36
- standard, 34
- standardization theorem, 35
- strongly  $\lambda$ -definable, 49
- strongly computable, 85
- strongly normalizable, 19
- strongly normalizing, 19
- subject construction lemma, 75
- subject expansion, 78
- subject reduction theorem, 77
- substitution, 7
- subterm, 3
- term, 2
- term models, 61
- term rewriting, 15
- terms, 73
- theory, 8
- total, 127
- total recursive functions, 44, 129
- trivial, 94
- Turing complete, 132
- Turing completeness, 49
- Turing machine, 127
- typable, 83
- type context, 72
- type substitutions, 92
- type variables, 72
- types, 72
- unary contexts, 7
- unifiable, 97
- unification, 97
- unifier, 97
- unifying type, 97
- union, 95
- unique normal form, 20
- universal Turing machine, 130
- unsolvable, 36
- valid, 2
- valuation, 59
- variable, 3
- variable convention, 6
- variable-domain, 95
- variable-range, 95
- weak extensionality rule, 57
- weak head reduction, 58
- weak reduction, 58
- weakening, 73
- weakly normalizing, 19
- well-defined, 92



# Bibliography

- [Bar84] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.
- [Bar92] H. P. Barendregt. Lambda calculi with types. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. OUP, 1992. Also downloadable from `ftp://ftp.cs.kun.nl/pub/CompMath.Found/HBKJ.ps.Z`.
- [BB94] H. P. Barendregt and H. Barendsen. Introduction to lambda calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988 revised 1994.
- [Böh68] C. Böhm. Alcune proprietà delle forme  $\beta$ - $\eta$ -normali nel  $\lambda$ - $K$  calcolo. *Pubblcazioni dell' Istituto per le Applicazioni del Calcolo*, 696, 1968.
- [CF58] H. B. Curry and R. Feys. *Combinatory Logic, Vol. I*. North-Holland, 1958.
- [Cut80] N. J. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.
- [dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Math.*, 34:381–392, 1972.
- [GLT89] J. Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.
- [Han94] C. Hankin. *Lambda Calculi: A Guide for Computer Scientists*, volume 3 of *Graduate Texts in Computer Science*. Oxford University Press, 1994.

- [Hin97] J. R. Hindley. *Basic Simple Type Theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1997.
- [HS80] J. R. Hindley and J. P. Seldin, editors. *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.
- [HS86] J. R. Hindley and J. P. Seldin, editors. *Introduction to Combinators and  $\lambda$ -calculus*. Cambridge University Press, 1986.
- [Plo72] G. D. Plotkin. A set-theoretical definition of application. Memorandum MIP-R-95, School of Artificial Intelligence, University of Edinburgh, 1972.
- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
- [Sch76] H. Schwichtenberg. Definierbare Funktionen im  $\lambda$ -Kalkül mit Typen. *Archiv Logik Grundlagenforsch*, 17:113–114, 1976.
- [Sel07] P. Selinger. *Lecture Notes on the Lambda Calculus*. 2001–2007.
- [Sta79] R. Statman. The typed  $\lambda$ -calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–81, 1979.
- [Tai67] W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32:198–212, 1967.
- [Tur42] A. M. Turing. Notes published in [HS80], 1942.