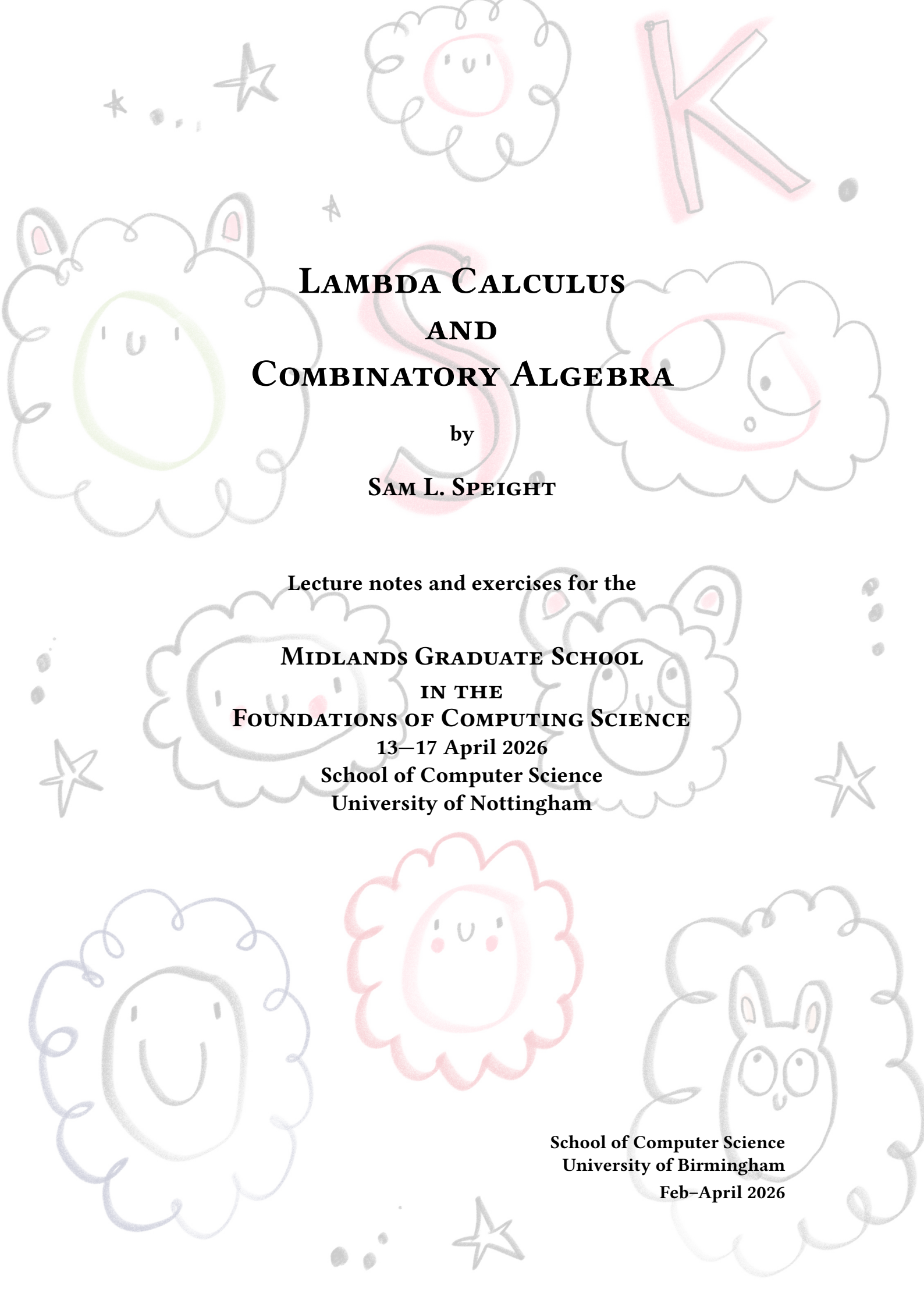

Lambda Calculus and Combinatory Algebra

by

Sam L. Speight

Lecture notes and exercises for the

Midlands Graduate School
in the
Foundations of Computing Science
13–17 April 2026
School of Computer Science
University of Nottingham

School of Computer Science
University of Birmingham
Feb–April 2026

Abstract

Lambda calculus and combinatory logic are both theories of *functions*—things that compute or *do*. They are formal systems in which we can program and perform logical reasoning. Superficially, they are a bit different—the main difference being that lambda calculus uses variable binding, whereas combinatory logic does not. Upon further investigation, they are pretty well—though not perfectly—aligned. We will concentrate on the untyped lambda calculus. Although variable binding can be fiddly, lambda calculus has a rather simple and intuitive syntax, which belies its power—both as a model of computation and as an idea. As well as its syntax and connection to combinatory logic, we will study the semantics of the lambda calculus via combinatory algebras. Lambda calculus (perhaps typed) is the heart of many modern functional programming languages and proof assistants.

Acknowledgements

Three primary sources of inspiration for these lecture notes were Andrew D. Ker's 2009 notes from the *Lambda Calculus and Types* course at Oxford [Ker09], Barendregt's textbook [Bar85] and Selinger's tutorial paper [Sel02].

Gratitude goes to Tom de Jong for the LaTeXtemplate and to Leah Riley for the illustration on the titlepage.

Please contact s.l.speight@bham.ac.uk with any comments or corrections.

Contents

Chapter 1

Very Brief History

The idea of combinators came about in the 1920s due to Schönfinkel, who had the intention of reformulating predicate logic without the need for variable binding [Sch24]. Later that decade, Curry rediscovered the idea whilst analyzing substitution. Fun fact: apparently, the idea of Currying also goes back to Schönfinkel; the name stuck despite Curry attributing the idea to Schönfinkel.

Lambda calculus came about later—in the 1930s—thanks to Church [Chu32]. Why 'λ'? In a letter, Church wrote that it was a modification of $\hat{x}$ to $\wedge x$ to λx, where the first of these is used by Russel and Whitehead in *Principia Mathematica* to denote class abstraction. Later in life, he is supposed to have said that λ was chosen essentially at random. For a more thorough history, see [CH09; Sel09].

Chapter 2

Syntax of Lambda Calculus

2.1 Terms

Terms of the λ-calculus are certain finite strings of symbols. We have a countably infinite set V of symbols called 'variables'. In addition, we have the symbols '(', ')', '.' and, of course, 'λ' (we could also allow constants, but choose not to).

The set Λ of terms (well-formed strings of symbols) is inductively generated by the following rules. This is to say, Λ is the smallest set of strings of symbols satisfying the following rules.

$$\frac{x \in V}{x \in \Lambda}\ \text{VAR} \qquad \frac{t \in \Lambda \qquad u \in \Lambda}{(tu) \in \Lambda}\ \text{APP} \qquad \frac{t \in \Lambda \qquad x \in V}{(\lambda x.t)}\ \text{ABS}$$

An upshot of this is that we may define functions out of Λ by *recursion* and prove things about all terms terms by *induction*. For both of these, there is one case for each of the above rules.

We adopt the following conventions:

- We drop parentheses with the understanding that application binds tighter than abstraction and application associates to the left. For example, $\lambda x.tuv$ is shorthand for $(\lambda x.((tu)v))$.
- Nested abstractions can be brought under a single λ. For example, $\lambda xy.t$ is shorthand for $\lambda x.\lambda y.t$.

2.2 Variable binding

The variable x in the term $\lambda x.t$ is *bound* by λ. Chances are you've come across bound variables before.

$$\sum_{i=0}^{n} i^2 \qquad \int_0^1 x^2 \,\mathrm{d}x \qquad \forall y.P(y)$$

In the summation on the left, the variable i is bound; in the integral in the middle, the variable x is bound; in the logical formula on the right, the variable y is bound.

The *free variables* of a λ-term t are those variables occurring in t that are not bound by a λ. Formally, the set $\mathsf{FV}(t)$ of free variables of a term t is defined by recursion on t.

- $\mathsf{FV}(x) \coloneqq \{x\}$
- $\mathsf{FV}(tu) \coloneqq \mathsf{FV}(t) \cup \mathsf{FV}(u)$
- $\mathsf{FV}(\lambda x.t) \coloneqq \mathsf{FV}(t) - \{x\}$

The final clause dictates that λ binds any occurrences of x in t. A closed term is a term t for which $\mathsf{FV}(t) = \emptyset$. Note that a variable may occur both free and bound in the same term, for example, x in $t \coloneqq x(\lambda x.x)$. $x \in \mathsf{FV}(t)$ because it occurs free in t (as well as occurring bound).

Bound variables are dummy variables in the sense that they can be renamed without changing the content of an expression. For example, the following two summations are the same.

$$\sum_{i=0}^{n} i^2 \qquad \sum_{j=0}^{n} j^2$$

In λ-calculus, we call two terms *α-equivalent* when they differ only in their bound variables. For example, the following two terms are α-equivalent.

$$\lambda x.xz \qquad \lambda y.yz$$

We consider α-equivalent terms to be syntactically identical, which is denoted using $\equiv$. We reserve $=$ for object-level equality, which is weaker.

Moreover, we adopt the 'Barendregt's variable convention':

> In a given mathematical context, we may assume that all bound variables are distinct from each other and from any free variables.

This is possible because terms are finite strings, yet we have a countably infinite supply of variables.

2.3 Substitution

We may substitute a term for a free occurrences of a variable in another term. The result $t[u/x]$ of substituting the term u for free occurrences of x in t is defined recursively as follows.

$$\frac{}{t = t}\ \text{REFL} \qquad \frac{t_1 = t_2}{t_2 = t_1}\ \text{SYM} \qquad \frac{t_1 = t_2 \qquad t_2 = t_3}{t_1 = t_3}\ \text{TRANS}$$

$$\frac{t_1 = t_2 \qquad u_1 = u_2}{t_1 u_1 = t_2 u_2}\ =\text{APP} \qquad \frac{t_1 = t_2}{\lambda x.t_1 = \lambda x.t_2}\ \xi$$

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

Figure 2.1: The rules for λ-theories

- $x[u/x] :\equiv u$
- $y[u/x] :\equiv y$ (where $y \not\equiv x$)
- $(t_1 t_2)[u/x] :\equiv (t_1[u/x])(t_2[u/x])$
- $(\lambda y.t)[u/x] :\equiv \lambda y.(t[u/x])$

Note that, in the final clause, we do not have to stipulate that $y \not\equiv x$ or $y \notin \mathsf{FV}(u)$ due to Barendregt's variable convention.

Exercise 2.1. Show by induction on s that $t_1[t_2/x][t_3/y] \equiv t_1[t_3/y][t_2[t_3/y]/x]$.

2.4 Lambda theories

Here we introduce equational theories of λ-calculus, or λ-theories. These are particular sets of equations between λ-terms. Such object-level equations are denoted using $=$ (formally, equations are pairs of λ-terms).

Definition 2.2. A λ-theory is a set of equations between λ-terms that is closed under the rules given in Figure 2.1. For a λ-theory T, write $T \vdash t = t'$ when $t = t' \in T$. The unique smallest λ-theory is called $\lambda\beta$.

Technically, each rule in Figure 2.1 specifies a family of pairs, whose first component is a set of equations (the premises), and whose second component is an equation (the conclusion). There may be 'side conditions' on rules (written above the line in parentheses), which cut down the family from what it would otherwise be.

The first row of rules ensure that $=$ is an equivalence relation. The second row of rules says that $=$ respects application and abstraction.

Finally, we have the β-rule. This gives life to λ-calculus. It tells us that abstraction forms functions, application is function application, and the action of a function on its argument is given by substitution. However, there is no formal distinction between functions and data that we feed to functions. For example, we may apply a function to itself! This oddity contributes to the power of the lambda calculus.

Exercise 2.3. Show that $\lambda\beta \vdash \lambda pq.(\lambda xy.y)pq = \lambda ab.a$ by drawing a proof tree.

Closed λ-terms are sometimes called 'combinators'.

Theorem 2.4 (First recursion theorem)**.** *Every λ-term has a fixed point. That is, for every λ-term f there is a λ-term u such that $\lambda\beta \vdash u = fu$.*

Proof. Consider 'Curry's paradoxical combinator': $\mathbf{y} :\equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. We have $\lambda\beta \vdash \mathbf{y}f = f(\mathbf{y}f)$ (exercise). This means that $\mathbf{y}f$ is a fixed point of f. □

Exercise 2.5. Verify that $\lambda\beta \vdash \mathbf{y}f = f(\mathbf{y}f)$.

We can add rules to those in Figure 2.1 to generate λ-theories. Equations can be considered as rules with no premises. For example, we have the η-equation:

$$\lambda x.tx = t \qquad (x \notin \mathsf{FV}(t)) \tag{η}$$

This makes every term a function. Moreover, we have the following extensionality rule.

$$\frac{t_1 x = t_2 x \qquad (x \notin \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2))}{t_1 = t_2}\ \text{EXT}$$

This equates two terms that produce the same result when applied to an arbitrary argument.

Exercise 2.6. (i) Show that $\lambda\beta + \eta = \lambda\beta + \{\lambda xy.xy = \lambda x.x\}$.

(ii) Show that $\lambda\beta + \eta = \lambda\beta + \text{EXT}$.

2.5 List of exercises

1. Exercise 2.1 on nested substitutions
2. Exercise 2.3 on proofs in $\lambda\beta$
3. Exercise 2.5 on Curry's paradoxical combinator yielding fixed points
4. Exercise 2.6 on equivalence of λ-theories

Chapter 3

Reduction

Computation is dynamic; it is a process. In this chapter, we will give dynamics to the lambda calculus by orienting the β-equation. Ultimately, this allows us to deduce consistency of the equational theory $\lambda\beta$.

3.1 Reduction relations

Given a binary relation R on Λ, we define another binary relation $\to_R$ on Λ—the '(one-step) R-reduction' relation—inductively.

$$\frac{(t_1, t_2) \in R}{t_1 \to_R t_2}\ R$$

$$\frac{t_1 \to_R t_2}{t_1 u \to_R t_2 u}\ \textsc{app-left} \qquad \frac{u_1 \to_R u_2}{t u_1 \to_R t u_2}\ \textsc{app-right}$$

$$\frac{t_1 \to_R t_2}{\lambda x.t_1 \to_R \lambda x.t_2}\ \textsc{abs}$$

While $\to_R$ is one step of R reduction, we can also consider many steps of R-reduction and R-equality:

- The reflexive transitive closure of $\to_R$ is denoted by $\twoheadrightarrow_R$. This is obtained by adding a reflexivity rule and a transitivity rule to those above.
- The reflexive symmetric transitive closure of $\to_R$ is denoted $=_R$. This is obtained by adding reflexivity, symmetry and transitivity rules.

3.1.1 Properties of reduction relations

The following notions make sense for any binary relation on a set. However, we are interested in these properties in the context of reduction relations $\to_R$ on Λ. All properties are parametrized by R, though this is sometimes left implicit if R is understood.

- A term $t \in \Lambda$ is in (or is an) *R-normal form* when there is no $t' \in \Lambda$ for which $t \to_R t'$.
- A term $t \in \Lambda$ *has an R-normal form* (or is *weakly R-normalizable*) when there is a normal form $t' \in \Lambda$ such that $t \twoheadrightarrow_R t'$.
- A term $t \in \Lambda$ is *strongly R-normalizable* when every sequence of reductions ends in an R-normal form.
- $\to_R$ is *weakly normalizing* when every term has an R-normal form.
- $\to_R$ is *strongly normalizing* when every term is strongly R-normalizable.
- $\to_R$ has the *diamond property* when $r \to_R s_1$ and $r \to_R s_2$ implies $s_1 \to_R t$ and $s_2 \to_R t$ for some t.
- $\to_R$ is *Church-Rosser* when $\twoheadrightarrow_R$ has the diamond property.

Exercise 3.1. Suppose $\to_R$ is Church-Rosser. If $t \in \Lambda$ has an R-normal form, then it is unique.

Proposition 3.2. *If $\to_R$ has the diamond property, then so does $\twoheadrightarrow_R$.*

Proof. By 'diagram chase'. □

Exercise 3.3. Suppose $\to_R$ is Church-Rosser. Show that $t_1 =_R t_2$ if and only if there is some t_3 such that $t_1 \twoheadrightarrow_R t_3$ and $t_2 \twoheadrightarrow_R t_3$. Hint: this is also a diagram chase.

Hence, if two terms are normalizable, then we can check equality by comparing normal forms.

3.2 Beta reduction

The relation we are most interested in is β-reduction, that is:

$$R \coloneqq \{((\lambda x.t)u, t[u/x])\}$$

The following is intuitively clear, but carrying out a formal proof should help solidify understanding.

Exercise 3.4. Show that $=_\beta$ is $\lambda\beta$.

So we can study the equational theory $\lambda\beta$ via the reduction relation $\to_\beta$.

Consider the combinator $\Omega \coloneqq (\lambda x.xx)(\lambda x.xx)$. Then there is only one possible β-reduction. That is: $\Omega \to_\beta \Omega$. From this we conclude that the lambda calculus is not even weakly normalizing. However, as we show in the next section, $\to_\beta$ is Church-Rosser. Hence, β-normal forms are unique.

Exercise 3.5. Consider Turing's fixed-point combinator: $\boldsymbol{\theta} \coloneqq (\lambda xy.y(xxy))(\lambda xy.y(xxy))$. Show that $f(\boldsymbol{\theta} f) \twoheadrightarrow_\beta \boldsymbol{\theta} f$. Show that not necessarily $f(\mathbf{y}f) \twoheadrightarrow_\beta \mathbf{y}f$.

3.3 Church-Rosser

Note that $\to_\beta$ does not have the diamond property: consider $(\lambda x.xx)((\lambda y.y)t)$.

Theorem 3.6 (Church-Rosser [CR36]). *$\to_\beta$ is Church-Rosser.*

The path to proving this result, which we tread in the remainder of this section, is due to Martin-Löf and Tait:

1. Define a notion $\to_{||}$ of 'parallel reduction';
2. Show that $\twoheadrightarrow_\beta$ is the reflexive transitive closure of $\to_{||}$;
3. Show that $\to_{||}$ has the diamond property;
4. Conclude that $\to_\beta$ is Church-Rosser.

The relation $\to_{||}$ of parallel reduction is inductively defined by the following rules.

$$\frac{}{t \to_{||} t}\ \textsc{refl}$$

$$\frac{t_1 \to_{||} t_2 \qquad u_1 \to_{||} u_2}{t_1 u_1 \to_{||} t_2 u_2}\ \textsc{app} \qquad \frac{t_1 \to_{||} t_2}{\lambda x.t_1 \to_{||} \lambda x.t_2}\ \textsc{abs}$$

$$\frac{t_1 \to_{||} t_2 \qquad u_1 \to_{||} u_2}{(\lambda x.t_1)u_1 \to_{||} t_2[u_2/x]}\ ||\beta$$

Exercise 3.7. Show that $\to_\beta \subseteq \to_{||} \subseteq \twoheadrightarrow_\beta$.

Proposition 3.8. *The reflexive transitive closure of $\to_{||}$ is $\twoheadrightarrow_\beta$.*

Proof. From Exercise 3.7 and the fact that taking the reflexive transitive closure of a relation is monotone with respect to $\subseteq$, we have $\twoheadrightarrow_\beta \subseteq \twoheadrightarrow_{||} \subseteq \twoheadrightarrow_\beta$. □

Proposition 3.9. *If $t \to_{||} t'$ and $u \to_{||} u'$, then $t[u/x] \to_{||} t'[u'/x]$.*

Proof. By induction on the derivation of $t \to_{||} t'$.

Case: REFL. As $t' \equiv t$, our goal reduces to showing $t[u/x] \to_{||} t[u'/x]$. We do a further induction on the structure of t.

Subcase: VAR. We have two cases: $t \equiv x$ and $t \equiv y$ for some variable $y \not\equiv x$. The first case amounts to showing $u \to_{||} u'$, which holds by assumption. The second case amounts to showing $y \to_{||} y$, which holds by REFL.

Subcase: $t = t_1 t_2$. This amounts to showing $t_1[u/x]t_2[u/x] \to_{||} t_1[u'/x]t_2[u'/x]$, which holds by the induction hypothesis.

Subcase: $t \equiv \lambda y.s$. Amounts to showing $\lambda y.(s[u/x]) \to_{||} \lambda y.(s[u'/x])$, which holds by induction hypothesis.

Case: APP. Exercise.

Case: ABS. Exercise.

Case: $||\beta$. We have to show that $((\lambda y.t_1)u_1)[u/x] \to_{||} (t_2[u_2/y])[u'/x]$, assuming $t_1 \to_{||} t_2$ and $u_1 \to_{||} u_2$. The induction hypothesis is $t_1[u/x] \to_{||} t_2[u'/x]$ and $u_1[u/x] \to_{||} u_2[u'/x]$. We calculate:

$$\begin{aligned} ((\lambda y.t_1)u_1)[u/x] &\equiv ((\lambda y.t_1)[u/x])(u_1[u/x]) \\ &\to_{||} t_2[u'/x][u_2[u'/x]/y] && \text{(by IH and } ||\beta) \\ &\to_{||} (t_2[u_2/y])[u'/x] && \text{(by Exercise 2.1)} \end{aligned}$$

□

Exercise 3.10. Fill in the APP and ABS cases from the proof of Proposition 3.9.

Lemma 3.11. *$\to_{||}$ has the diamond property.*

Proof. We show that $r \to_R s_1$ and $r \to_R s_2$ implies $s_1 \to_R t$ and $s_2 \to_R t$ for some t by induction on the derivation of $r \to_R s_1$. We just give two cases.

Case: REFL. As $s_1 \equiv r$, take $t \equiv s_2$.

Case: APP. There are two cases.

Subcase 1. The last rules in the derivations of $r \to_R s_1$ and $r \to_R s_2$ are respectively:

$$\frac{p \to_{||} p_1 \qquad q \to_{||} q_1}{pq \to_{||} p_1 q_1}\ \text{APP} \qquad \frac{p \to_{||} p_2 \qquad q \to_{||} q_2}{pq \to_{||} p_2 q_2}\ \text{APP}$$

Then by the induction hypothesis, we get t_1, t_2 such that $p_1 \to_{||} t_1$, $p_2 \to_{||} t_1$, $q_1 \to_{||} t_2$ and $q_2 \to_{||} t_2$. Take $t \equiv t_1 t_2$. Then $p_1 q_1 \to_{||} t$ and $p_2 q_2 \to_{||} t$ by the APP rule, as required.

Subcase 2. The last rules in the derivations of $r \to_R s_1$ and $r \to_R s_2$ are respectively:

$$\frac{\lambda x.p \to_{||} \lambda x.p_1 \qquad q \to_{||} q_1}{(\lambda x.p)q \to_{||} (\lambda x.p_1)q_1}\ \text{APP} \qquad \frac{p \to_{||} p_2 \qquad q \to_{||} q_2}{(\lambda x.p)q \to_{||} p_2[q_2/x]}\ ||\beta$$

Then we must have $p \to_{||} p_1$. So by the induction hypothesis, we get t_1, t_2 such that $p_1 \to_{||} t_1$, $p_2 \to_{||} t_1$, $q_1 \to_{||} t_2$ and $q_2 \to_{||} t_2$. Take $t \equiv t_1[t_2/x]$. Then $s_1 \equiv (\lambda x.p)q \to_{||} t$ by the $||\beta$ rule, and $s_2 \equiv p_2[q_2/x] \to_{||} t$ by Proposition 3.9. □

Exercise 3.12. Complete the remaining cases in the proof of Lemma 3.11.

This completes the proof of Theorem 3.6, that $\to_\beta$ is Church-Rosser.

3.4 Consistency

We turn our focus back to λ-theories. What does it mean for a λ-theory to be consistent or inconsistent? We don't have any (native) connectives like $\neg$ (negation) and $\wedge$ (and). Well, we should be able to deduce interesting consequences from a consistent theory. That is, consistent theories should be informative. Turning this around, an inconcsistent theory should be completely uninformative. This idea is formalized in the following definition.

Definition 3.13. A λ-theory T is inconsistent when for all $t, t' \in \Lambda$, $T \vdash t = t'$. A λ-theory T is consistent when it is not inconsistent, ie. there exist terms t, t' such that $T \nvdash t = t'$.

In the next chapter, we will encode Booleans (true and false) as λ-terms. Another reasonable definition of inconsistency is that a theory equates true and false. We will see that this is equivalent to the above definition.

Theorem 3.14. *$\lambda\beta$ is consistent.*

Proof. Take any two distinct normal forms s, s'. Then $\lambda\beta \vdash s = s'$ iff $s =_\beta s'$ iff (by Exercise 3.3) there is some t such that $s \twoheadrightarrow_\beta$ and $s' \twoheadrightarrow_\beta t$. But there is no such t, as s, s' are in normal form. □

Terms without a normal form are computations that never terminate. Can we lump all such terms together (in an 'undefined' category)?

Theorem 3.15. *$T_{\mathsf{NF}} \coloneqq \lambda\beta + \{t = t' \mid t, t'$ closed and not weakly normalizable$\}$ is inconsistent.*

Proof. If r is a closed term, then the following are closed terms.

$$\lambda xy.xr \qquad \lambda xy.yr$$

are in normal form. Then for arbitrary s, s':

$$\begin{aligned} T_{\mathsf{NF}} \vdash s &= (\lambda xy.x)sr) \\ &= (\lambda xy.xr)((\lambda xy.x)s)((\lambda xy.x)s') \\ &= (\lambda xy.yr)((\lambda xy.x)s)((\lambda xy.x)s') \\ &= (\lambda xy.x)s'r \\ &= s' \end{aligned}$$

□

3.5 List of exercises

1. Exercise 3.1 on Church-Rosser and unique normal forms
2. Exercise 3.3 on equality and reduction

3. Exercise 3.4 on the relationship between β-reduction and the equational theory $\lambda\beta$
4. Exercise 3.5 on Turing's fixed-point combinator
5. Exercise 3.7 on inclusion of reduction relations
6. Exercise 3.10 on parallel reduction and substitution
7. Exercise 3.12 on the diamond property for $\rightarrow_{||}$

Chapter 4

Computability and Undecidability

In this section, we study the notions of computability and undecidability in lambda calculus. We relate lambda calculus to total recursive functions, a model of (total) computation due to Kleene. We also show that there is no algorithm that decides whether two λ-terms are equated in $\lambda\beta$. In this section, all equalities are in $\lambda\beta$ (so hold in any λ-theory).

4.1 Coding

Traditionally, when we talk about computability, we mean computability of number-theoretic functions, that is, functions $\mathbb{N}^m \to \mathbb{N}$. Therefore, we need a way to talk about numbers in lambda calculus. It turns out that we can encode many objects in lambda calculus, as well as operations on these objects.

4.1.1 Booleans

Let's first look at Booleans, ie. true and false.

$$\mathbf{t} :\equiv \lambda xy.x \qquad \mathbf{f} :\equiv \lambda xy.y$$

(We've already being using these combinators.) So 'true' is a function that takes two arguments and returns the first (ignoring the second), whereas 'false' is a function that takes two arguments and returns the second (ignoring the first).

We've described our Booleans as functions. So what's the justification for calling them Booleans? Well, we can condition on these Booleans.

$$\mathbf{cond} :\equiv \lambda xyz.zxy$$

Look what happens when we feed $t_1, t_2, b \in \Lambda$ to **cond**, where b is one of our Booleans.

$$\mathbf{cond}t_1t_2\mathbf{t} \equiv (\lambda xyz.zxy)t_1t_2\mathbf{t} = (\lambda xy.x)t_1t_2 = t_1$$
$$\mathbf{cond}t_1t_2\mathbf{f} \equiv (\lambda xyz.zxy)t_1t_2\mathbf{f} = (\lambda xy.y)t_1t_2 = t_2$$

Exercise 4.1. Find a combinator **and** that implements conjunction of truth values.

Recall that we defined a λ-theory to be inconsistent when it equates all terms. In fact, it is enough to equate $\mathbf{t}$ and $\mathbf{f}$.

Proposition 4.2. $\mathbf{t} = \mathbf{f}$ *if any only if for all* $t_1, t_2 \in \Lambda$, $t_1 = t_2$.

Proof. The backwards direction is trivial. For the forwards direction, remember that $\mathbf{t}t_1t_2 = t_1$. So if $\mathbf{t} = \mathbf{f}$, then $\mathbf{f}t_1t_2 = t_1$. But also $\mathbf{f}t_1t_2 = t_2$. So $t_1 = t_2$. □

4.1.2 Church numerals

How about numbers?

$$\overline{0} :\equiv \lambda fx.x \qquad \overline{n+1} :\equiv \lambda fx.f(\overline{n}fx)$$

Let's calculate:

$$\begin{aligned}
&\overline{1} \equiv \lambda fx.f(\overline{0}fx) \equiv \lambda fx.f((\lambda fx.x)fx) = \lambda fx.fx \\
&\overline{2} \equiv \lambda fx.f(\overline{1}fx) = \lambda fx.f((\lambda fx.fx)fx) = \lambda fx.f(fx) \\
&\overline{3} \equiv \lambda fx.f(\overline{2}fx) = \lambda fx.f((\lambda fx.ffx)fx = \lambda fx.f(f(fx)) \\
&\ldots
\end{aligned}$$

So the n^{th} Church numeral has n 'f's in its body.

So to define a successor function, we just need to find a way to add another f into the body of a Church numeral.

$$\mathbf{succ} :\equiv \lambda nfx.f(nfx)$$

This indeed has the property that

$$\mathbf{succ}\overline{n} = \overline{n+1}$$

Another useful function is

$$\mathbf{zero?} :\equiv \lambda n.n(\lambda x.\mathbf{f})\mathbf{t}$$

Let's calculate:

$$\begin{aligned}
&\mathbf{zero?}\overline{0} = (\lambda fx.x)(\lambda x.\mathbf{f})\mathbf{t} = \mathbf{t} \\
&\mathbf{zero?}\overline{n+1} = (\lambda fx.f(\overline{n}fx))(\lambda x.\mathbf{f})\mathbf{t} = (\lambda x.\mathbf{f})(\overline{n}(\lambda x.\mathbf{f})\mathbf{t}) = \mathbf{f}
\end{aligned}$$

So **zero?** tests for zero.

4.2 Definability of total recursive functions

Definition 4.3. A function $\phi : \mathbb{N}^m \to \mathbb{N}$ is **λ-definable** when there exists $f \in \Lambda$ such that for every $(n_1, ..., n_m) \in \mathbb{N}^m$ we have:

$$f\overline{n_1}...\overline{n_m} = \overline{\phi(n_1, ..., n_m)}$$

That is, f computes on Church numerals in the same way that ϕ computes on external natural numbers.

Kleene defined the total recursive functions as a model of total computation. The set of total recursive functions $\mathbb{N}^m \to \mathbb{N}$ is defined inductively by the following clauses.

- The constantly zero function $n \mapsto 0$ is total recursive.
- The successor function $n \mapsto n + 1$ is total recursive.
- The projection functions $(n_1, ..., n_m) \mapsto n_i$ for $1 \leq i \leq m$ are total recursive.
- Closure under composition: if $\phi : \mathbb{N}^m \to \mathbb{N}$ is total recursive, and $\psi_i : \mathbb{N}^l \to \mathbb{N}$ $(1 \leq i \leq m)$ are total recursive, then

  $$\phi \circ (\psi_1, ..., \psi_m) : (n_1, ..., n_l) \mapsto \phi(\psi_1(n_1, ..., n_l), ..., \phi_m(n_1, ..., n_l))$$

  is total recursive.
- Closure under primitive recursion: if $\alpha : \mathbb{N}^m \to \mathbb{N}$ is total recursive and $\psi : \mathbb{N}^{m+2} \to \mathbb{N}$ is total recursive, then the function $\phi : \mathbb{N}^{m+1} \to \mathbb{N}$ defined below is total recursive.

  $$\phi(k, n_1, ..., n_m) \coloneqq \begin{cases} \alpha(n_1, ..., n_m) & \text{if } k = 0 \\ \psi\,(\phi(k-1, n_1, ..., n_m), k-1, n_1, ..., n_m) & \text{if } k > 0 \end{cases}$$

  See how ψ is allowed to make a 'recursive call'. This process terminates when $k = 0$.
- Closure under minimization: if $\psi : \mathbb{N}^{m+1} \to \mathbb{N}$ is total recursive and there exists some $k \in \mathbb{N}$ such that $\psi(k, n_1, ..., n_m) = 0$, then $\phi : (n_1, ..., n_m) \mapsto k$, where k is the least natural number such that $\psi(k, n_1, ..., n_m) = 0$, is total recursive.

  ϕ is guaranteed to be a total function because of the condition on ψ.

Theorem 4.4. *Every total recursive function is λ-definable.*

Proof. We'll just do a couple of cases. The successor function is definable by **succ**. For minimization, suppose ψ is definable by f. Define:

$$g :\equiv \mathbf{y}(\lambda zyx_1, ..., x_m.\mathbf{zero?}(fyx_1...x_m)y(z(\mathbf{succ}y)x_1...x_m))$$

Then:

$$g = \lambda yx_1, ..., x_m.\mathbf{zero?}(fyx_1...x_m)y(g(\mathbf{succ}y)x_1...x_m)$$

As ψ is definable by f, we get:

$$g\overline{k}\overline{n_1}...\overline{n_m} = \begin{cases} \overline{k} & \text{if } \psi(k, n_1, ..., n_m) = 0 \\ \overline{g\overline{k+1}\overline{n_1}...\overline{n_m}} & \text{otherwise} \end{cases}$$

The function ϕ is definable by $h :\equiv g\overline{0}$. When we feed $\overline{n_1}...\overline{n_m}$ to h, it returns $\overline{0}$ if $\psi(0, n_1, ..., n_m) = 0$. If not, we increment the first argument given to g. As start from zero and go up, h returns the Church numeral of the first k for which $\psi(k, n_1, ..., n_m) = 0$. □

Exercise 4.5. Complete the remaining cases in the proof of Theorem 4.4. For closure under primitive recursion, you may take for granted a combinator $\mathbf{rcase} \in \Lambda$ such that

$$\mathbf{rcase}\overline{n}fg = \begin{cases} f & \text{if } n = 0 \\ g\overline{n-1} & \text{if } n > 0 \end{cases}$$

The converse to this theorem also holds. That is, every λ-definable function is total recursive. We won't prove this. A formal proof requires an effective Gödel numbering of λ-terms, that is, a bijection $\# : \Lambda \longrightarrow \mathbb{N}$ such that operations on λ-terms are definable by total recursive functions on natural numbers. In the next section, we will use the fact that an effective Gödel numbering exists in order to prove an undecidability result.

An analogue of Theorem 4.4 holds for partial recursive functions too. In fact, the set of partial recursive functions, the set of λ-definable functions and the set of functions computable by some Turing machine all coincide. Functions in this class are said to be 'Turing-computable'. The notion of Turing-computability is remarkably robust: many models of computation agree about the set of computable number-theoretic functions (though not necessarily higher-order functions [LN15]).

Finally, we can replace equality in the definition of λ-definable with $\longrightarrow_\beta$ (though we have to use $\boldsymbol{\theta}$ instead of $\mathbf{y}$, cf. Exercise 3.5).

4.3 Undecidability of $\lambda\beta$

Decidability questions ask if there is an algorithm for determining whether objects are in a given set or not. As with computability, the domain is the natural numbers. A subset $S \subseteq \mathbb{N}$ is decidable if there is a total recursive function ϕ such that:

$$\phi(n) = \begin{cases} 0 & \text{if } n \in S \\ 1 & \text{if } n \notin S \end{cases}$$

Let $\# : \Lambda \xrightarrow{\sim} \mathbb{N}$ be an effective Gödel numbering of λ-terms. This means, in particular, that we have functions $\mathrm{app} : \mathbb{N}^2 \longrightarrow \mathbb{N}$ and $\mathrm{gnum} : \mathbb{N} \longrightarrow \mathbb{N}$ such that

$$\mathrm{app}(\#(t), \#(u)) = \#(tu) \qquad \mathrm{gnum}(n) = \#(\overline{n})$$

The operation $\ulcorner - \urcorner := \overline{\#(-)}$ encodes the syntax of λ-calculus in λ-calculus! By Theorem 4.4, we have λ-terms $\mathbf{app}$ and $\mathbf{gnum}$ with:

$$\mathbf{app}\ \ulcorner t \urcorner\ \ulcorner u \urcorner = \ulcorner tu \urcorner \qquad \mathbf{gnum}\ \ulcorner t \urcorner = \ulcorner\ulcorner t \urcorner\urcorner$$

Theorem 4.6 (Second recursion theorem). *For all $f \in \Lambda$, there is some $u \in \Lambda$ such that $f \ulcorner u \urcorner = u$.*

Proof. Given f, define

$$t :\equiv \lambda x.f(\mathbf{app}x(\mathbf{gnum}x)) \qquad u :\equiv t\ulcorner t\urcorner$$

Now calculate:

$$u \equiv t\ulcorner t\urcorner = f(\mathbf{app}\ulcorner t\urcorner(\mathbf{gnum}\ulcorner t\urcorner)) = f(\mathbf{app}\ulcorner t\urcorner\ulcorner t\urcorner) = f\ulcorner t\ulcorner t\urcorner\urcorner = f\ulcorner u\urcorner$$

□

Theorem 4.7 (Scott-Curry theorem). *Let A and B be two non-empty sets of λ-terms closed under $=_\beta$. There is no term f such that, for all s, $f\ulcorner s\urcorner = \overline{0}$ or $f\ulcorner s\urcorner = \overline{1}$, additionally with*

$$f\ulcorner u\urcorner = \begin{cases} \overline{0} & \text{if } u \in A \\ \overline{1} & \text{otherwise} \end{cases}$$

Proof. Assume A and B are disjoint (otherwise the result is trivial—what should f return for u in the intersection?). Suppose we have an f as in the theorem statement. Let $a \in A$ and $b \in B$. Define:

$$g :\equiv \lambda x.\mathbf{cond}ba(\mathbf{zero?}fx)$$

Then by the Second recursion theorem, there is some u such that

$$u = g\ulcorner u\urcorner = \begin{cases} b & \text{if } f\ulcorner u\urcorner = \overline{0} \\ a & \text{if } f\ulcorner u\urcorner = \overline{1} \end{cases}$$

Either $f\ulcorner u\urcorner = \overline{0}$ or $f\ulcorner u\urcorner = \overline{1}$ by assumption. WLOG suppose that $f\ulcorner u\urcorner = \overline{0}$; then $u = b$; so $u \in B$; so $f\ulcorner u\urcorner = \overline{1}$; so $\overline{0} = \overline{1}$; contradiction ($\overline{0}$ and $\overline{1}$ are distinct normal forms, so $\lambda\beta$ doesn't equate them). □

Corollary 4.8 (Undecidability of equality in $\lambda\beta$). *Let S be a non-empty proper subset of Λ that is closed under equality. Then $\#(S) \subset \mathbb{N}$ is not decidable.*

Proof. If $\#(S)$ were decidable, then by Theorem 4.4 there would be a λ-term contradicting Theorem 4.7 instantiated with $\#(S)$ and $\mathbb{N} - \#(S)$. □

4.4 List of exercises

1. Exercise 4.1 on conjunction of Booleans
2. Exercise 4.5 on λ-definability

Chapter 5

Combinatory Algebra

5.1 Combinatory logic

1. Combinatory logic
2. Translation of λ-terms to combinatory terms
3. Remark that combinatory logic also defines all total recursive functions
4. Combinatory completeness

5.2 Combinatory algebras

1. Equivalence of **SK** and combinatory completeness
2. Algebraically pathological
3. Interpreting combinatory logic and lambda calculus
4. Failure of soundness

5.3 Lambda algebras

1. Soundness and completeness for pure lambda calculus
2. Equational characterizations
3. Failure of soundness for general λ-theories

5.4 Examples of lambda algebras

1. Reflexive CCC models
2. Graph models

Recall that a category $\mathbb{C}$ is cartesian closed when it has finite products and exponentials.

Definition 5.1. An object $U \in \mathbb{C}$ is **reflexive** when it comes equipped with morphisms $\mathsf{lam} : U^U \to U$ and $\mathsf{app} : U \to U^U$ such that $\mathsf{app} \circ \mathsf{lam} = \mathsf{id}_{U^U}$. A **reflexive cartesian closed category** is a cartesian closed category together with a reflexive object.

5.5 List of exercises

Chapter 6

Variations on Combinatory Algebras

There are many variations on the theme of combinatory algebras. Each variation is a model of some kind of combinatory logic and the associated notion of computation. In this chapter, we do a whistle-stop tour of some of these variations. In particular, for the variations of combinatory algebras considered, we give:

1. the relevant kind of applicative structure;
2. the definition in terms of combinators;
3. the equivalent definition in terms of combinatory completeness.

6.1 Typed combinatory algebras

Types constrain the behaviour of terms; they prevent us from doing crazy things like applying a function to itself. While the untyped lambda calculus is not strongly normalizing, typed lambda calculus is. Under the Curry-Howard correspondence, types are logical propositions, or, more generally, mathematical objects (like $\mathbb{N}$ and $\mathsf{List}(\mathbb{N})$).

Definition 6.1. A **type structure** $\mathbb{T}$ is a set of 'types' closed under two binary operations $\to$ and $\times$.

Definition 6.2. A **typed applicative structure** is a type structure $\mathbb{T}$ together with a family of sets A_T indexed by types and a family of binary operations $\bullet_{S,T} : A_{S\to T} \times A_S \to A_T$ indexed by pairs (S, T) of types.

Definition 6.3 (Longley [Lon99])**.** A typed applicative structure is a **typed combinatory algebra** when...

Definition 6.4. A typed applicative structure is **combinatory complete** when...

Theorem 6.5. *A typed applicative structure is a typed combinatory algebra iff it is combinatory complete.*

We can recover (untyped) combinatory algebras from typed combinatory algebras provided that we have a 'universal type', that is, a type $U \in \mathbb{T}$ together with families of combinators $\mathbf{lam}_T \in A_{T \to U}$ and $\mathbf{app}_T : A_{U \to T}$ such that $\forall x \in A_T.\ \mathbf{app}_T(\mathbf{lam}_T(x))$. Then we can make A_U into a combinatory algebra.

6.2 Partial combinatory algebras

Partiality (or non-termination) is a central notion in computability. We have encountered non-normalizing λ-terms. Generally speaking, if computation is the process of mechanically following a finite set of instructions, while the set of instructions must be finite, we may never reach a state where where we are not asked to follow a further instruction. Another justification for the centrality of partiality is that it is possible to define a universal partial computable function, though not a universal total computable function.

Definition 6.6. A **partial applicative structure** is a set A together with a partial binary operation $\bullet : A \times A \rightharpoonup A$.

Definition 6.7. A **partial combinatory algebra** is a partial applicative structure admitting combinators $\mathbf{S}, \mathbf{K} \in A$ satisfying:

$$\forall ab \in A.\ \mathbf{K}ab = a$$
$$\forall ab \in A.\ \mathbf{S}ab\downarrow \qquad\qquad \forall abc \in A.\ \mathbf{S}abc \simeq ac(bc)$$

$\mathbf{K}ab = a$ implies that the applications on the left are defined. The notation $\mathbf{S}ab\downarrow$ means that the application $\mathbf{S}ab$ is defined. Moreover, $\mathbf{S}abc \simeq ac(bc)$ means that either both sides are undefined or else both sides are defined and equal.

Definition 6.8. A partial applicative structure is **combinatorially complete** when...

Theorem 6.9. *A partial applicative structure is a partial combinatory algebra iff it is combinatory complete.*

Example 6.10. The prototypical example of a partial combinatory algebra is 'Kleene's first algebra' $\mathcal{K}_1$.

Realizability models based on $\mathcal{K}_1$ satisfy (formal) Church's thesis: every function $\mathbb{N} \to \mathbb{N}$ is computable.

'Computational effects' refer to features of programs over and above their functional (input-output) behaviour. Partiality is one of a variety of computational effects that may be considered in relation to combinatory algebras; other instances are nondeterminism and state [CAT19]. Recently, a general approach has emerged, namely 'monadic combinatory algebras' [CGKM25].

Combinator identity	Logical principle
$\mathbf{B}abc = a(bc)$	cut/composition
$\mathbf{C}abc = acb$	exchange
$\mathbf{I}a = a$	identity
$\mathbf{K}a!b = a$	weakening
$\mathbf{W}a!b = a!b!b$	contraction
$\mathbf{F}!a!b = !(ab)$	functoriality
$\mathbf{D}!a = a$	counit/dereliction
$\delta!a = !!a$	comultiplication

Table 6.13: Combinator identities for linear combinatory algebras, together with their corresponding logical principle

6.3 Linear combinatory algebras

Linear logic is a 'resource sensitive' interpretation of logic. For example, we are used to being able to deduce the proposition 'p and p' from the proposition 'p', as well as deduce 'p' from 'p and q'. However, we cannot do this under the linear logic interpretation of 'and' (technically, the 'multiplicative and'). If we think of propositions p and q as resources, then we are not allowed to unrestrictedly duplicate or discard resources. Some resources may be duplicable and discardable, but they must be marked as such by a 'modality' !.

Definition 6.11. A **linear applicative structure** is a set A together with a binary operation $\bullet : A \times A \to A$ and a unary operation $! : A \to A$.

Definition 6.12 (Abramsky [AHS02])**.** A **linear combinatory algebra** is a linear applicative structure admitting combinators $\mathbf{B}, \mathbf{C}, \mathbf{I}, \mathbf{K}, \mathbf{W}, \mathbf{F}, \mathbf{D}, \delta \in A$ satisfying the identities in Table 6.13.

The combinator identities in Table 6.13 come with their corresponding logical (or categorical) principle. Note that the behaviour of **W** and **K** are controlled by !, that is, we can only duplicate and discard arguments appearing under a !.

Definition 6.14. A linear applicative structure is combinatory complete when...

Theorem 6.15. *A linear applicative structure is a linear combinatory algebra iff it is combinatory complete.*

Bibliography

[AHS02] Samson Abramsky, Esfandiar Haghverdi, and Philip Scott. "Geometry of Interaction and linear combinatory algebras". In: *Mathematical Structures in Computer Science* 12.5 (2002), pp. 625–665. DOI: 10.1017/S0960129502003730 (cit. on p. 21).

[Bar85] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*. Vol. 103. Studies in logic and the foundations of mathematics. North-Holland, 1985. ISBN: 978-0-444-86748-3 (cit. on p. ii).

[CAT19] Liron Cohen, Sofia Abreu Faro, and Ross Tate. "The Effects of Effects on Constructivism". In: *Electronic Notes in Theoretical Computer Science* 347 (2019). Proceedings of the Thirty-Fifth Conference on the Mathematical Foundations of Programming Semantics, pp. 87–120. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2019.09.006. URL: https://www.sciencedirect.com/science/article/pii/S1571066119301227 (cit. on p. 20).

[CGKM25] Liron Cohen, Ariel Grunfeld, Dominik Kirst, and Étienne Miquey. "From Partial to Monadic: Combinatory Algebra with Effects". In: *10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025)*. Ed. by Maribel Fernández. Vol. 337. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, 14:1–14:22. ISBN: 978-3-95977-374-4. DOI: 10.4230/LIPIcs.FSCD.2025.14. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2025.14 (cit. on p. 20).

[CH09] Felice Cardone and J. Roger Hindley. "Lambda-Calculus and Combinators in the 20th Century". In: *Logic from Russell to Church*. Ed. by Dov M. Gabbay and John Woods. Vol. 5. Handbook of the History of Logic. North-Holland, 2009, pp. 723–817. DOI: https://doi.org/10.1016/S1874-5857(09)70018-4. URL: https://www.sciencedirect.com/science/article/pii/S1874585709700184 (cit. on p. 1).

[Chu32] Alonzo Church. "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics* 33.2 (1932), pp. 346–366. ISSN: 0003486X, 19398980. URL: http://www.jstor.org/stable/1968337 (visited on 04/13/2026) (cit. on p. 1).

[CR36] Alonzo Church and J. B. Rosser. "Some properties of conversion". In: *Transactions of the American Mathematical Society* 39.3 (1936), pp. 472–482. DOI: 10.1090/S0002-9947-1936-1501858-0 (cit. on p. 8).

[Ker09] Andrew D. Ker. *Lambda Calculus and Types*. Lecture notes, University of Oxford. 2009. URL: https://www.cs.ox.ac.uk/andrew.ker/docs/lambdacalculus-lecture-notes-ht2009.pdf (cit. on p. ii).

[LN15] John Longley and Dag Normann. *Higher-Order Computability*. 1st. Springer Publishing Company, Incorporated, 2015. ISBN: 3662479915 (cit. on p. 15).

[Lon99] John Longley. *Unifying Typed and Untyped Realizability*. Available online. 1999. URL: http://homepages.inf.ed.ac.uk/jrl/Research/unifying.txt (cit. on p. 19).

[Sch24] M. Schönfinkel. "Über die Bausteine der mathematischen Logik". In: *Mathematische Annalen* 92 (1924), pp. 305–316. URL: http://eudml.org/doc/159074 (cit. on p. 1).

[Sel02] Peter Selinger. "The lambda calculus is algebraic". In: *Journal of Functional Programming* 12.6 (2002), pp. 549–566. DOI: 10.1017/S0956796801004294 (cit. on p. ii).

[Sel09] Jonathan P. Seldin. "The Logic of Church and Curry". In: *Logic from Russell to Church*. Ed. by Dov M. Gabbay and John Woods. Vol. 5. Handbook of the History of Logic. North-Holland, 2009, pp. 819–873. DOI: https://doi.org/10.1016/S1874-5857(09)70019-6. URL: https://www.sciencedirect.com/science/article/pii/S1874585709700196 (cit. on p. 1).