Constructive Type Theory

Notes for Marktoberdorf 1997

Peter Aczel
Manchester

July 12, 1997

Preface

These notes do not give a fully accurate representation of the planned lectures. Nevertheless they should provide useful background material which I hope will supplement the bibliography. The notes focus only on non-dependent type theory, which will be the topic of the first few lectures. The remaining lectures will extend the material to encompass constructive type theory with dependent types.

The first part of these notes looks at aspects of the non-dependent part of constructive type theory from the foundational point of view, starting with a quote from Godel's Dialectica paper. The second part of these notes consists of material prepared for an MSc course at Manchester University on the simply typed lambda calculus. It includes sections on Deduction and the untyped lambda calculus. Unfortunately I have not had time to adjust the presentation to properly serve the purposes of the Summer School. Nevertheless I hope that it may prove useful as a metatheoretical counterpart to the foundational first part.

Contents

Abstract

In recent years a considerable variety of type theories have been introduced, investigated and used in computer science, for a variety of purposes; e.g. the design of type systems for programming languages, program development and verification, formalised proofs and the foundations of mathematics and computing.

In these lectures I will give an introduction to a particular variety of dependent type theory that has its origins in proof theory and the foundations of constructive mathematics. Its main distinctive feature is its disciplined open-ended use of a direct semantics consisting of Per Martin-Lof's 'meaning explanations'. It makes full use of the 'propositions as types' paradigm to represent logic and gets its strength and expressiveness through the use of inductive data types and a hierarchy of type universes. In contrast the proof theoretically stronger, but impredicative dependent type theories such as the calculus of constructions can represent logic and inductive data types using impredicative second order quantification.

The main emphasis of my lectures will be on the development of an intuitive understanding of the ideas that are incorporated into constructive type theory and how informal abstract mathematics can be represented in the type theory. I will use examples from algebra of machine checked formalisation using the LEGO computer system.

Bibliography

1. M. Beeson, "Foundations of Constructive Mathematics", Springer-Verlag, 1985.

2. E. Bishop and D. Bridges, "Constructive Analysis", Springer-Verlag, 1985.

3. J. Girard, Y. Lafont and P. Taylor, "Proofs and Types", Cambridge Tracts in Theoretical Computer Science 7, Cambridge University Press, 1989.

4. J. Lambek and P.J. Scott, "Introduction to higher order categorical logic", Cambridge Studies in Advanced Mathematics 7, Cambridge University Press, 1986.

5. Z. Luo, "Computation and Reasoning: A Type Theory for Computer Science", Oxford University Press, 1994.

6. Z. Luo and R. Pollack, "LEGO Proof Development System: User's Manual", ECS-LFCS-92-211, LFCS Report Series, CS Department, Edinburgh, 1992.

7. P. Martin-Lof, "Intuitionistic Type Theory", Bibliopolis, Napoli, 1984.

8. B. Nordstrom, K. Petersson and J. Smith, "Programming in Martin-Lof's Type Theory, An Inroduction", Monographs on Computer Science 7, Oxford University Press, 1990.

9. B. Nordstrom and K. Petersson and J. Smith, "Martin-Lof's Type Theory." A chapter in Handbook of Logic in Computer Science, written in 1994, to appear. (Available by ftp from ftp://ftp.cs.chalmers.se/pub/cs-reports/papers/smith/hlcs.ps.gz)

10. A. Ranta, "Type Theoretical Grammar", Oxford University Press, 1994.

11. S. Thompson, "Type Theory and Functional Programming", Addison Wesley, 1991.

12. A. S. Troelstra and D. van Dalen, "Constructivism in Mathematics, An Introduction", Vols. 1, 2, Studies in Logic, North-Holland, 1988.

13. A. S. Troelstra and H. Schwichtenberg, Basic Proof Theory", Cambridge Tracts in Theoretical Computer Science 43, Cambridge University Press, 1996.

Part I: The Constructive notion of Type

1 Some Background and Initial Explanations

1.1 Some Background

I start with an extended quotation from Godel's important, but not sufficiently well known Dialectica paper[1]. In my opinion this quotation conveys many of the key ideas involved in the constructive notion of type.

> '... one can prove the consistency of number-theory by using the notion of computable functions of finite type over the natural numbers, together with certain very elementary principle for constructing these functions. We explain the notion of 'a computable function of type t' as follows:
>
> 1. The computable functions of type 0 are the natural numbers.
> 2. If the notions of 'computable function of type t_0', 'computable function of type t_1',...,'computable function of type t_k', (for some $k \geq 1$) have been defined, then a computable function of type $(t_0, t_1, \ldots, t_k)$ is defined to be an everywhere effective (and thus constructively recognisable) operation which correlates a computable function of type t_0 with each k-tuple of computable functions of types $t_1, \ldots, t_k$.
>
> We consider that this notion [2] can be directly understood[3] by a person who already understands the notion of 'computable function of type t_i' ($i = 1, \ldots, k$). Regard the type t as a variable, and we get the notion of 'computable function of finite type t'; this is the notion we need for the consistency proof.

[1] *'On a Hitherto Unexploited Extension of the Finitary Standpoint'*, in Dialectica 12 (1958), pp. 280-287. I use a translation by W. Hodges. The footnotes 2-5 are Godel's.

[2] One may doubt whether we have a sharp enough characterisation of this notion, but not whether it satisfies the axioms given below. We meet the same apparent paradox with a notion which lies at the base of intuitionistic logic, viz. the notion of a proof being 'correct in content'. As shown by the following considerations and the intuitionistically interpreted theory of recursive functions and functionals, within certain limits these two notions can replace each other as basic notion. Note that because the notion of a computable function mustn't implicitly contain the notion of proof, the effectivity of the operations must be immediately apparent from the string of definitions - which is the case with all the functions of the system T given below.

[3] As is well known, A.M. Turing used the notion of a computer in order to give a definition of the notion of a first-order computable function. But if this latter notion wasn't already intelligible, then the question whether Turing's definition is adequate would be meaningless.

> Besides the axioms of identity (including those for functions[4]), Peano's third and fourth axioms, and the rule of substitution for free variables, the only obvious axioms we need are those that allow us
>
> (i) to define functions by setting them equal to terms constructed out of variables and previously defined constants, and simple induction on one number variable; and
>
> (ii) to use inference by complete induction on one number variable.
>
> Thus the axioms of this system (call it T) are formally much the same[5] as those of primitive recursive number-theory, except that the variables (apart from those on which induction is performed) and the defined constants can be of any finite types over the natural numbers.'

Some Historical Steps The following is a somewhat miscellaneous list of what seem to me to be some of the key steps in the development of the ideas of constructive type theory. I have mentioned some specific names, but in a more carefully prepared list there would no doubt be several other names and topics to add.

1. Frege's syntactically based distinction between function and object and his classification of functions into the number and kind of the argument places leading to his hierarchy of levels of functions.

2. Russell's predicative theory of types with its ramified hierachy.

3. Ramsey's 'simplification' of Russell's full theory, with the impredicative 'Axiom of Reducability', into an impredicative simple type theory.

4. Church's simple type theory, the simply typed lambda calculus.

5. The Church/Curry untyped lambda calculus and combinatory logic.

6. The Brouwer/Heyting/Kolmogorov explanations of Intuitionistic Logic.

7. Godel's Dialectica Interpretation.

[4]Functions are taken to be identical if and only if they are intensionally identical, i.e. if and only if they have the same definition.

[5]In a definition where a function is set equal to a term there is this difference: we can define a function P of higher type by writing

$$[P(x_1, x_2, \ldots, x_n)](y_1, y_2, \ldots, y_m) = E.$$

But this difference disappears if we replace many-place functions by one-place functions, in the manner of A. Church.

8. The Gentzen/Prawitz analysis of logic: natural deduction and normalisation of proofs, sequent calculus and cut-elimination.
9. De Bruijn's Automath.
10. Kreisel/Goodman theories of constructions.
11. Dana Scott's 'Constructive Validity' paper.
12. The Curry-Howard correspondence between formulae and types.
13. Bishop's Constructive Mathematics.
14. Martin-Lof's dependent type theories and his meaning explanations.
15. Girard's system F and the Coquand/Huet calculus of constructions.
16. Categorical Logic.
17. Computer implementations of functional programming languages, logics and type theories.

In thinking about and presenting type theories it is necessary to distinguish between two complementary approaches.

- Foundational
- Metatheoretical

In the Foundational approach the aim is to set up formal systems for codifying the practise of constructive mathematics and to elaborate a framework of pre-mathematical explanations which can be used to give a direct interpretation of the formal systems. By contrast the metatheoretical approach investigates the proof theory and semantics of formal systems of type theory, within the standard framework of accepted mathematical practise - perhaps as codified in classical axiomatic set theory.

I take both these approaches to be important, one complementing the other. Both approaches will be considered in these lectures. We start by taking the foundational approach with initial explanations of the notion of a type and the type of functions from one type to another.

1.2 Initial Explanations

1.2.1 The Notion of Type

The question: What is a type? has several variants that it is worthwhile to consider at the same time.

- What does it mean that something is a type?
- What do we know when we know that something is a type?
- What do we need to do to define a type?

The ontological slant of the original question is replaced by semantical, epistemological and ... slants in the variants. So an answer to the question can also have variants with various slants. Here are two versions of an answer.

- A type is explained by saying what an object of the type is and when objects of the type are equal (identical, the same).
- A type is defined by stating what we have to do in order to construct an object of the type and what conditions need to be satisfied for such objects to be equal.

The notion of equality between the objects of a type has to be an equivalence; i.e. it has to be reflexive, symmetric and transitive.

The Forms of Judgment Given a type A we understand the form of judgment

$$a \text{ is an object of type } A,$$

and, for objects a, a' of type A, the form of judgment

$$a, a' \text{ are equal objects of type } A$$

These forms are abbreviated

$$a : A$$

and

$$a = a' : A.$$

The following rules are clearly justified by the requirements on equality.

$$\frac{a : A}{a = a : A} \qquad \frac{a = a' : A}{a' = a : A} \qquad \frac{a = a' : A \quad a' = a'' : A}{a = a'' : A}$$

1.2.2 The Raw Syntax

The objects of constructive type theory will be represented by expressions E whose abstract raw syntax is given by the following syntax equation.

$$E ::= x \mid c \mid E(E) \mid [x]E$$

Here x stands for **variables**, c for **constants**, $E_1(E_2)$ will represent **function application** and $[x]E$ will represent **function abstraction**. This syntax is a variation on the syntax for the untyped lambda calculus. The abstraction operation $E \mapsto [x]E$ is variable binding, so that (free) occurrences of x in E get bound in $[x]E$. Expressions will be identified up to α-convertibility. Because bound variables can be relabelled without changing the expression there is no notion of bound variable. We say that an expression **depends** on the variables that occur (free) in it. We assume that substitution (for free occurrences of variables, up to α-convertibility) is defined in one of the standard ways. The result of a simultaneous substitution will be written

$$E[x_1 := E_1, \ldots, x_n := E_n].$$

Associated with abstraction are the β-equations

$$([x]E)(E') = E[x := E'].$$

These equations, along with defining equations for the defined constants determine an operational semantics for the evaluation of expressions.

We will use various kinds of abbreviation in order to make expressions more readable. In particular a repeated application expression $E(E_1)\cdots(E_n)$ will be abbreviated $E(E_1, \ldots, E_n)$ and a repeated abstraction expression $[x_1]\cdots[x_n]E$ will be abbreviated $[x_1, \ldots, x_n]E$.

The constants will be classified into the **primitive constants** and the **defined constants**, the latter being further classified into the **explicitly defined constants** and the **implicitly defined constants**. The primitive constants simply evaluate to themselves. Each explicitly defined constant c is introduced with a defining equation

$$c = E$$

where E is a closed (variable free) expression that only involves previously introduced constants. The explicitly defined constant c will evaluate to whatever value the expression E evaluates to. The implicitly defined constants may have several defining equations, each having on the left the constant applied to a pattern of arguments involving primitive constants and variables. Pattern matching is used in determining the operational semantics. Also recursion is allowed; i.e. in a defining equation for an implicit constant, the constant may appear on the right hand side. Some discipline is needed in introducing implicit constants with their defining equations and operational semantics. But we do not try to give any general formulation for how this works. In particular cases it will hopefully be clear.

1.2.3 The Notion of Function

We assume given types A and B and we want to answer the question:

What is a function from A to B?

Our answer will be in terms of the operation of application of a function to an argument to yield a value.

> We state that f is a function from A to B if, when we apply f to an object a of type A we get an object $f(a)$ of type B and when we apply f to equal objects a, a' of type A we get equal objects $f(a), f(a')$ of type B.

What is the type of functions from A to B? It is clear from the above what the objects of this type are. But to have a type we also need to answer the question:

When are functions from A to B equal?

> Functions f, f' from A to B are equal if, whenever thay are applied to an arbitrary object a of type A we get equal objects $f(a), f(a')$ of type B.

By using the fact that equality on B is an equivalence we easily see that equality of functions from A to B is also an equivalence. So we have explained the type of functions from A to B. We will write $A \Rightarrow B$ for this type.[6]

The following rules can now be justified.

$$\frac{f : A \Rightarrow B \quad a : A}{f(a) : B} \qquad \frac{f : A \Rightarrow B \quad a = a' : A}{f(a) = f(a') : B} \qquad \frac{f = f' : A \Rightarrow B \quad a : A}{f(a) = f'(a) : B}$$

Examples Let us introduce the explicitly defined constant I with the defining equation

$$I = [x]x.$$

Then, for any expression E that has a value the expression $I(E)$ will have the same value. Then, clearly, for each type A we are justified in making the judgment

$$I : A \Rightarrow A$$

and can justify the rule

$$\frac{a : A}{I(a) = a : A}.$$

In a similar way we can define functions of several arguments. So, for example, we have the function K with defining equation

$$K = [x, y]x.$$

[6] The Gothenberg group have used $(A)B$ for this type. But we prefer to use a more familiar arrow notation.

Given types A, B, we can make the judgment

$$K : A \Rightarrow (B \Rightarrow A)$$

and justify the rule

$$\frac{a : A \qquad b : B}{K(a)(b) = a : A}.$$

As a third example we have S with defining equation

$$S = [x, y, z]x(z, y(z))$$

so that, given types A, B, C, we can make the judgment

$$S : ((A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$$

and justify the rule

$$\frac{f : A \Rightarrow (B \Rightarrow C) \qquad g : A \Rightarrow B \qquad a : A}{S(f)(g)(a) = f(a)(g(a)) : C}.$$

In future we shall use the standard abbreviatory convention that allows us to write $A \Rightarrow B \Rightarrow C$ rather than $A \Rightarrow (B \Rightarrow C)$, etc. So the typing judgment for S can be rewritten

$$S : (A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C.$$

1.2.4 Hypothetical Judgments with one Assumption

In general a judgment can involve a context of assumptions. So far we have only considered judgments with no assumptions; i.e. with an empty context. We now consider judgments with the context of a single assumption

$$x : A.$$

Here A is a type and x is a variable that is declared to be of type A by the assumption. We wish to explain, given a type B, the form of judgment

b is an object of type B under the assumption $x : A$

and, for such objects b, b' the form of judgment

b, b' are equal objects of type B under the assumption $x : A$.

We abbreviate these forms as

$$[x : A]\ b : B$$

and

$$[x : A]\ b = b' : B.$$

- We may make the judgment $[x : A]\ b : B$ if b is an expression that may depend on the variable x, but on no other variables, and we can justify the two rules

$$\frac{a : A}{b[x := a] : B} \qquad \frac{a = a' : A}{b[x := a] = b[x := a'] : B}.$$

- Given that $[x : A]\ b : B$ and $[x : A]\ b' : B$ we may make the judgment $[x : A]\ b = b' : B$ if we can justify the rule

$$\frac{a : A}{b[x := a] = b'[x := a] : B}.$$

Given these explanations we can clearly justify the following three rules.

$$\frac{[x : A]\ b : B \qquad a : A}{b[x := a] : B} \qquad \frac{[x : A]\ b : B \qquad a = a' : A}{b[x := a] = b[x := a'] : B},$$

$$\frac{[x : A]\ b = b' : B \qquad a : A}{b[x := a] = b'[x := a] : B}.$$

In addition we can justify the following extensionality rule for functions.

$$\frac{f : A \Rightarrow B \qquad f' : A \Rightarrow B \qquad [x : A]\ f(x) = f'(x) : B}{f = f' : A \Rightarrow B}$$

1.2.5 Function Abstraction

We have the following rules for function abstraction, given types A, B.

$$\frac{[x : A]\ b : B}{[x]b : A \Rightarrow B} \qquad \frac{[x : A]\ b : B \qquad a : A}{([x]b)(a) = b[x := a] : B}$$

Also, the following rules can be derived.

$$\frac{f : A \Rightarrow B}{[x](f(x)) = f : A \Rightarrow B}$$

$$\frac{[x : A]\ b = b' : B}{[x]b = [x]b' : A \Rightarrow B}$$

1.2.6 Hypothetical Judgments with any number of Assumptions

The general form of judgment that we want to consider here can have a context Γ of assumptions that will have the form

$$(*) \qquad x_1 : A_1, \ldots, x_n : A_n$$

with any number $n \geq 0$ of assumptions, where $A_1, \ldots, A_n$ are types. The variables $x_1, \ldots x_n$ must be pairwise distinct. We want to explain the form of judgment

$$[\Gamma]\ b : B$$

where Γ is a context $(*)$ as above and B is a type, and also, given $[\Gamma]\ b : B$ and $[\Gamma]\ b' : B$ the form of judgment

$$[\Gamma]\ b = b' : B.$$

These explanations are given by induction on the length n of the context Γ, the induction step generalising the previous explanations for the case when $n = 1$. So, given explanations for the forms of judgment, as above, with contexts Γ of a particular length n, explanations can be given for the forms of judgment

$$[x : A, \Gamma]\ b : B$$

and

$$[x : A, \Gamma]\ b = b' : B.$$

Rather than write out these explanations we just present the following rules that the explanations justify.

$$\frac{[x : A, \Gamma]\ b : B \qquad a : A}{[\Gamma]\ b[x := a] : B} \qquad \frac{[x : A, \Gamma]\ b : B \qquad a = a' : A}{[\Gamma]\ b[x := a] = b[x := a'] : B},$$

$$\frac{[x : A, \Gamma]\ b = b' : B \qquad a : A}{[\Gamma]\ b[x := a] = b'[x := a] : B}.$$

1.2.7 Relativisation of Rules

Having explained the general forms of hypothetical judgment we could go on to formulate and justify relative forms of all our rules. For example, given any context Δ and a type A we can justify the rule

$$\frac{[\Delta]\ a : A}{[\Delta]\ a = a : A}$$

which is the relative form of the reflexivity rule we have already considered, that rule being the special case when Δ is the empty context. We take it that the reader could fill in the routine details that would be needed.

1.3 Data Types

1.3.1 General Explanation of the notion of a data type

A **data type** is a type that is given to us in a particular kind of way. A data type is specified by giving the **introduction rules** for inductively generating the **canonical values** of the data type; i.e. those objects of the data type that are in **canonical form**. Each object in canonical form is built from its components using a **constructor**. Given the introduction rules for a data type A, we can give the explanation of A as a type.

> We know that a is an object of type A if we know how to evaluate a to a canonical value of type A. Objects of type A are equal if they evaluate to equal canonical values; i.e. canonical values that use the same constructor and whose components are equal, component by component.

Associated with each data type will be a **selector**, given as an implicitly defined constant, whose defining equations express the induction/recursion principle for the data type.

1.3.2 The data type of natural numbers

The data type '$\mathbb{N}$' of natural numbers has the constructors 0 and s with the introduction rules

$$0 : \text{`}\mathbb{N}\text{'} \qquad\qquad \frac{n : \text{`}\mathbb{N}\text{'}}{s(n) : \text{`}\mathbb{N}\text{'}}$$

The zero and successor objects 0 and $s(n)$, with n : '$\mathbb{N}$', are the canonical natural numbers. We know that a is a natural number if we know how to evaluate a to one of these canonical natural numbers. In particular a successor canonical natural number has the form $s(n)$ where n must evaluate to a previously generated natural number. Equality for canonical natural numbers is inductively generated using the rules

$$0 = 0 : \text{`}\mathbb{N}\text{'} \qquad\qquad \frac{n = n' : \text{`}\mathbb{N}\text{'}}{s(n) = s(n') : \text{`}\mathbb{N}\text{'}}$$

The selector natrec for the data type '$\mathbb{N}$' has the following defining equations for primitive recursion.

$$\begin{array}{ll} \mathsf{natrec}(d, e, 0) & = d \\ \mathsf{natrec}(d, e, s(n)) & = e(n, \mathsf{natrec}(d, e, n)) \end{array}$$

For each data type C we can justifiably make the judgement

$$\mathsf{natrec} : C \Rightarrow (\text{`}\mathbb{N}\text{'} \Rightarrow C \Rightarrow C) \Rightarrow \text{`}\mathbb{N}\text{'} \Rightarrow C$$

and justify the rules

$$\frac{d : C \qquad e : \text{‘}\mathbb{N}\text{’} \Rightarrow C \Rightarrow C}{\mathsf{natrec}(d, e, 0) = d : C}$$

and

$$\frac{d : C \qquad e : \text{‘}\mathbb{N}\text{’} \Rightarrow C \Rightarrow C \qquad n : \text{‘}\mathbb{N}\text{’}}{\mathsf{natrec}(d, e, s(n)) = e(n, \mathsf{natrec}(d, e, n)) : C}.$$

We can use natrec to define addition and multiplication on natural numbers. We introduce the explicitly defined constants with defining equations

$$\begin{aligned} add &= [m, n]\mathsf{natrec}(n, [x, y]s(y), m) \\ mult &= [m, n]\mathsf{natrec}(0, [x, y]add(y, n), m) \end{aligned}$$

Part II: Some Metatheory

2 Deduction

2.1 Inference Systems

2.1.1 The Definition

An **inference system** consists of

- a set of **statements**, $S, S', \ldots$
- a set of **steps**

$$\frac{S_1 \cdots S_n}{S}$$

with $n \geq 0$ **premisses** $S_1, \ldots, S_n$ and **conclusion** S being statements. When $n = 0$ then S is an **axiom**.

Given an inference system a **tree-proof** is an upward growing finite tree, labelled with statements, such that at each node, labelled with a statement S there is a step with conclusion S whose premisses label the children of the node. A tree-proof is a **tree-proof of** S if the statement S labels the root of the tree. If S has a tree-proof in an inference system Σ then S is a **theorem of** Σ, written $\Sigma \vdash S$, or just $\vdash S$ when Σ is understood from the context.

Exercises 2.1

1. *Show that the theorems of an inference system form the smallest set X of statements such that, for each step, if the premisses are in X then so is the conclusion.*

2. *Show that the set of proof-trees of an inference system form the smallest set Y of finite trees labelled with statements such that for each step*

$$\frac{S_1 \cdots S_n}{S}$$

if $\tau_1, \ldots, \tau_n$ are trees in Y whose roots are labelled with $S_1, \ldots, S_n$ respectively then τ is a tree in Y, where τ is the finite tree whose root is labelled with S that has the trees $\tau_1, \ldots, \tau_n$ as immediate subtrees.

3. [**Linear Proofs**]
Given an inference system a **linear proof of a statement** *S is a finite sequence $S_1, \ldots, S_m$ of statements, with $S_m = S$, such that for $i = 1, \ldots, m$ there is a step*

$$\frac{S_{i_1} \cdots S_{i_n}}{S_i}$$

with $1 \leq i_1, \ldots, i_n < i$.

Show that a statement is a theorem of the inference system iff it has a linear proof.

2.1.2 Formal Systems

A **formal system** is an inference system for which there is a data type T such that the following holds.

- Every statement can be represented by a value in T.
- There is an algorithm to determine whether or not a value in T represents a statement.
- There is an algorithm to determine whether or not a pair of values of T represent the same statement.
- there is an algorithm to determine whether or not a configuration

$$\frac{v_1 \cdots v_n}{v}$$

  of values in T represents a step of the inference system.

Note the following facts

- Given a formal system there is an algorithm to determine whether or not a finite tree of values of the data type represents a proof-tree or not.
- If Σ is a formal system then so is $\Sigma, S_1, \ldots, S_n$.

2.1.3 Rules of Inference

Usually the steps of an inference sytem are given by **rules**, each rule determining a set of steps called the **instances** of the rule. Often each rule is given **schematically** as a scheme; i.e. a configuration

$$\frac{\theta_1 \cdots \theta_n}{\theta}$$

of expressions involving **metavariables** that can be substituted for. The instances of the rule are then obtained by suitably substituting for the metavariables. What is meant by a suitable substitution has to be specified by an implicit or explicit **side condition** of the scheme. An **axiom scheme** is a special case of a schematically given rule in which all instances are steps having no premisses, so that the conclusion of each instance is an axiom.

Often a formal system is given by finitely many schematic rules, each having a syntactically specified matching algorithm for deciding whether or not a configuration of data values is an instance of the rule; i.e. can be obtained from the scheme by a suitable substitution for the metavariables.

2.1.4 Adding extra axioms

If $S_1, \ldots, S_n$ are statements of an inference system Σ then we write $\Sigma, S_1, \ldots, S_n$ for the inference system obtained from Σ by adding new axioms $S_1, \ldots S_n$; i.e. there are new steps

$$\frac{}{S_1} \quad \cdots \quad \frac{}{S_n}.$$

If Σ is understood then we just write

$$S_1, \ldots, S_n \vdash S$$

for $\Sigma, S_1, \ldots, S_n \vdash S$.

2.1.5 Semantics for Inference Systems

Often an inference syetm will have a **semantics**. For our purposes we will take a semantics for an inference system to be a collection of **interpretations**, I, each specifying the **correct** (or **true**) statements of the interpretation. We write $I \models S$ if S is a correct statement of I. An interpretation is **sound** if, for every step, if the premisses are correct then so is the conclusion. A semantics is sound if every interpretation is sound.

Given a semantics for an inference system, we write

$$S_1, \ldots S_n \models S$$

if for every interpretation in which $S_1, \ldots, S_n$ are correct the statement S is also correct. The semantics is **complete** if

$$S_1, \ldots S_n \models S \text{ implies } S_1, \ldots S_n \vdash S$$

and is **weakly complete** if

$$\models S \text{ implies } \vdash S.$$

Note that we can formulate a notion of strong completeness by defining '$X \vdash S$' and '$X \models S$', where X is a possibly infinite set of statements.

Exercise 2.2 *Show that if a semantics is sound then*

$$S_1, \ldots S_n \vdash S \textit{ implies } S_1, \ldots S_n \models S.$$

2.2 Intuitionistic Implication

We assume given a set of **atomic formulae** $A_0, \ldots$. The **formulae** are generated from the atomic formulae using the rule

$$\frac{A_1 \quad A_2}{(A_1 \to A_2)}$$

i.e. the rule that if A_1, A_2 are formulae then so is $(A_1 \to A_2)$.

Abbreviation Conventions

1. Leave out outermost parentheses.
2. Associate to the right.

so, for example

$$A_1 \to A_2 \to A_3 \to A_4$$

abbreviates

$$(A_1 \to (A_2 \to (A_3 \to A_4))).$$

2.2.1 A Hilbert-style formal system, H

We describe a formal system, H, whose statements are the formulae defined above. There are two axiom schemes **(K), (S)** and the rule of inference **(MP)** of *Modus Ponens*

Axiom Schemes

(K) $A \to B \to A$.

(S) $(A \to B \to C) \to (A \to B) \to A \to C$.

(MP) $\dfrac{A \to B \quad A}{B}$.

Each axiom scheme represents infinitely many instances; e.g. for each pair of formulae A, B the formula $A \to B \to A$ is an instance of $(Ax1)$.

A tree-proof of $A \to A$: Let A be any formula and let B be $A \to A$.

$$\dfrac{\dfrac{\overset{(S)}{(A \to B \to A) \to (A \to B) \to B} \qquad \overset{(K)}{A \to B \to A}}{(A \to B) \to B}\,(MP) \qquad \overset{(K)}{A \to B}}{B}\,(MP)$$

Note the conventions that we use to make the tree-proof more readable. Each leaf is given the name of the axiom scheme it is an instance of and each application of (MP) has been named.

A linear proof of $A \to A$: Again let A be any formula and let B be $A \to A$. Here is a linear proof consisting of five numbered formulae.

1.	$(A \to B \to A) \to (A \to B) \to B$	(S)
2.	$A \to B \to A$	(K)
3.	$(A \to B) \to B$	$MP(1,2)$
4.	$A \to B$	(K)
5.	B	$MP(3,4)$

Note some obvious conventions used on the right hand side of each line to justify that line.

Note that both the tree-proof and the linear proof are schematic - they apply uniformly to any formula A.

Exercise 2.3 (Deduction Theorem) *Show that if $A_1, \ldots A_n, A \vdash B$ then $A_1, \ldots, A_n \vdash A \to B$. [This is a standard result whose proof method can be found in many textbooks on logic.]*

2.2.2 Natural Deduction

The *natural* way to prove an implication $A \to B$ is to assume A and try to deduce B, making use of the assumption A, when needed. Once B has been successfully deduced from A then we can infer $A \to B$. Notice that although A was assumed, when trying to deduce B the assumption of A is dropped once the inference step has been made to get $A \to B$. This kind of inference step involves the **discharge** of the assumption A. This is a new idea, that is not part of the apparatus of inference systems we have been using so far. Nevertheless there is a convenient way to modify the notion of a tree-proof so as to allow for the discharge of assumptions.

In an inference system Σ a tree-proof from assumptions $S_1, \ldots, S_n$ is simply a tree-proof in the inference system $\Sigma, S_1, \ldots, S_n$ so that, at the leaves of the tree any of the assumptions $S_1, \ldots, S_n$ may appear as well as any axioms of Σ. In natural deduction tree-proofs we allow the possibility of discharging an assumption at an inference step. We will do this by labelling the occurrences of the assumption being discharged and at the same time labelling the inference step where the discharge happens with the same label. Of course the same label should not be used for different examples of assumption discharge in the same tree.

We can now describe the two rules for constructing ND tree-proofs for Intuitionistic Implication.

The **Introduction Rule for Implication**, abbreviated $(\to I)$, states that given an ND tree-proof τ of a formula B we can form an ND tree-proof of $A \to B$ having $A \to B$ at the root and having the tree τ as its only immediate subtree. But any undischarged occurrences of A as an assumption in τ can become discharged in the new tree.

The **Elimination Rule for Implication**, abbreviated $(\to E)$, is simply the familiar modus ponens rule that we have called (MP).

These rules are written schematically as follows.

$$(\to I)\quad \frac{\begin{array}{c}[A]\\ B\end{array}}{A \to B} \qquad\qquad (\to E)\quad \frac{A \to B \quad A}{B}$$

Note that in $(\to I)$ the assumption A has been placed in square brackets above B to indicate that it may be assumed in the proof of B, but then discharged at the inference step.

Examples of ND tree-proofs We give ND tree-proofs of the axioms of our Hilbert-style formal system for Intuitionistic Implication.

(K)

$$\dfrac{\dfrac{\overset{1}{A}}{B \to A}\,(\to I)}{A \to B \to A}\,(\to I)[1]$$

(S)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overset{1}{A \to B \to C} \quad \overset{3}{A}}{B \to C}\,(\to E) \qquad \dfrac{\overset{2}{A \to B} \quad \overset{3}{A}}{B}\,(\to E)}{C}\,(\to E)}{A \to C}\,(\to I)[3]}{(A \to B) \to A \to C}\,(\to I)[2]}{(A \to B \to C) \to (A \to B) \to A \to C}\,(\to I)[1]$$

Examples of ND linear proofs Here are two linear ND proofs, where we have used some obvious conventions to indicate where an assumption is made and where it gets discharged.

(K)

$$\begin{array}{lll}
1. & A & (ass) \\
2. & B & (ass) \\
3. & B \to A & (\to I)[2](1) \\
4. & A \to B \to A & (\to I)[1](3)
\end{array}$$

(S)

1.	$A \to B \to C$	(ass)
2.	$A \to B$	(ass)
3.	A	(ass)
4.	$B \to C$	$(\to E)(1,3)$
5.	B	$(\to E)(2,3)$
6.	C	$(\to E)(4,5)$
7.	$A \to C$	$(\to I)[3](6)$
8.	$(A \to B) \to A \to C$	$(\to I)[2](7)$
9.	$(A \to B \to C) \to (A \to B) \to A \to C$	$(\to I)[1](8)$

Proofs of $A \to A$

$$\frac{\overset{1}{A}}{A \to A}\ (\to I)[1]$$

1.	A	(ass)
2.	$A \to A$	$(\to I)[1](1)$

2.2.3 Sequent Formulation, ND, of Natural Deduction

In this formulation of natural deduction the use of a special kind of tree-proof is replaced by the use of **sequents** as the statements of a formal system having standard Hilbert-style rules of inference. A **sequent** has the form

$$\Gamma \mathrel{|\!\blacktriangleright} A$$

where Γ is a finite sequence $A_1, \ldots, A_n$ of formulae and A is a formula.
Note: In the literature other symbols are often used, instead of $\mathrel{|\!\blacktriangleright}$; e.g. '$\vdash$' or '$\Rightarrow$'. But these symbols are often also used in other ways, leading to ambiguity.

The formal system is specified by the axiom scheme (ass) and the two rules of inference $(\to I)$ and $(\to E)$.

$$(ass) \quad \Gamma \mathrel{|\!\blacktriangleright} A \quad (A \text{ in } \Gamma)$$

$$(\to I) \quad \frac{\Gamma, A \mathrel{|\!\blacktriangleright} B}{\Gamma \mathrel{|\!\blacktriangleright} A \to B}$$

$$(\to E) \quad \frac{\Gamma \mathrel{|\!\blacktriangleright} A \to B \qquad \Gamma \mathrel{|\!\blacktriangleright} A}{\Gamma \mathrel{|\!\blacktriangleright} B}$$

Write $\Gamma \mathrel{|\!\blacktriangleright}_{ND} A$ if $\Gamma \mathrel{|\!\blacktriangleright} A$ is a theorem of this formal system; i.e. if $ND \vdash \Gamma \mathrel{|\!\blacktriangleright} A$.

Theorem 2.4 *$\Gamma \vdash\!\blacktriangleright_{ND} A$ iff there is an ND tree-proof of A, whose undischarged assumptions are all in Γ.*

Write $\Gamma \vdash_H A$ if $\Gamma \vdash A$ in the Hilbert style inference system for Implication with the axiom schemes (K), (S) and the rule (MP).

Theorem 2.5 *$\Gamma \vdash\!\blacktriangleright_{ND} A$ iff $\Gamma \vdash_H A$.*

2.2.4 Normal ND tree-proofs

A **redex** in an ND tree-proof is an occurrence of an instance of the rule $(\to E)$ whose left hand premiss is the conclusion of an instance of the rule $(\to I)$; i.e. a part of the tree of the form

$$\begin{matrix} \dfrac{\dfrac{\begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{A \to B}\,(\to I) \qquad \begin{matrix}\vdots\\ A\end{matrix}}{B}\,(\to E) \\ \vdots \end{matrix}$$

An ND tree-proof is **normal** if it contains no redex. Note that a redex is a detour and it is natural to consider 'simplifying' a redex by replacing it by its **contractum**, which is the ND tree-proof obtained from the ND tree-proof of B, appearing in the redex, by replacing all the occurrences of A, as an assumption that gets discharged in the redex, by the ND tree-proof of A occuring in the redex. The result may be pictured

$$\begin{matrix}\vdots\\ A\\ \vdots\\ B\end{matrix}$$

Theorem 2.6 *If $\Gamma \vdash_{ND} A$ then there is a normal ND tree-proof of A whose undischarged assumptions are all in Γ.*

Proof Idea: Given an ND tree-proof, if it is not already normal then choose a redex and replace it by its contractum. Do this repeatedly, until no more redexes remain. It is not obvious that this procedure eventually terminates successfully. Nevertheless it turns out that however the redexes to be contracted are chosen the procedure will indeed terminate. This is the **strong normalisation property** for Natural Deduction. In fact it turns out to be easier to show that a particular systematic strategy for choosing redexes leads to termination. This is the (weak) **normalisation property**. We will return to complete this proof later, when we discuss the simply typed lambda calculus.

2.2.5 Sequent Calculus SC

This formal system uses the same sequents as statements that are used in ND, but has slightly different rules. First of all there is the structural rule

$$\frac{\Gamma \mathrel{|\!\blacktriangleright} A}{\Gamma' \mathrel{|\!\blacktriangleright} A}$$

with the side condition that the same formulae appear in Γ as are in Γ'. The effect of having this rule is that the order that assumptions appear in the sequence Γ is irrelevent, as is the number of times the same formula appears in the sequence; i.e. Γ can be treated as a *set* of assumptions. The remaining rules are

$$\begin{array}{ll}
(ass) & \Gamma \mathrel{|\!\blacktriangleright} A \quad (A \text{ in } \Gamma) \\[2ex]
(\rightarrow R) & \dfrac{\Gamma, A \mathrel{|\!\blacktriangleright} B}{\Gamma \mathrel{|\!\blacktriangleright} A \rightarrow B} \\[2ex]
(\rightarrow L) & \dfrac{\Gamma \mathrel{|\!\blacktriangleright} A \qquad \Gamma, B \mathrel{|\!\blacktriangleright} C}{\Gamma, A \rightarrow B \mathrel{|\!\blacktriangleright} C} \\[2ex]
(cut) & \dfrac{\Gamma, A \mathrel{|\!\blacktriangleright} C \qquad \Gamma \mathrel{|\!\blacktriangleright} A}{\Gamma \mathrel{|\!\blacktriangleright} C}
\end{array}$$

We write $\Gamma \mathrel{|\!\blacktriangleright}_{SC} A$ if $SC \vdash \Gamma \mathrel{|\!\blacktriangleright} A$.

Theorem 2.7 *$\Gamma \mathrel{|\!\blacktriangleright}_{SC} A$ iff $\Gamma \mathrel{|\!\blacktriangleright}_{ND} A$.*

Let SC^- be SC with the cut rule left out.

Theorem 2.8 *If $\Gamma \mathrel{|\!\blacktriangleright}_{SC} A$ then $\Gamma \mathrel{|\!\blacktriangleright}_{SC^-} A$.*

This is the **Cut Elimination Theorem** for this sequent calculus. It is also called Gentzen's **Hauptsatz**. Gentzen gave an algorithm for systematically removing cuts from a tree-proof in SC. By theorems 3 and 4 it suffices to show that if there is a normal ND tree-proof of A whose undischarged assumptions are all in Γ then $\Gamma \mathrel{|\!\blacktriangleright}_{SC^-} A$. This is not hard to do by induction on the size of the normal ND tree-proof.

Note that Natural Deduction and the Sequent Calculus are closely related approaches to deduction in which normalisation of ND tree-proofs corresponds to cut-elimination. We will prefer to focus on the natural deduction approach.

Exercise 2.9 *Prove theorems 1,2,4,5, using theorem 3 in the proof of theorem 5.*

2.3 Intuitionistic Propositional Logic

In this section we extend the purely implicational logic of the previous section by adding the other logical constants of intuitionistic propositional logic; i.e. we add the two binary connectives $\wedge$ and $\vee$ for conjunction and disjunction and the special constant $\perp$ for absurdity. The other propositional constants $\leftrightarrow$ and $\neg$ of bi-implication and negation can be defined in the standard way. So every non-atomic formula has one of the forms

$$(A_1 \to A_2), \quad (A_1 \wedge A_2), \quad (A_1 \vee A_2), \quad \perp$$

and we use the definitions

$$(A_1 \leftrightarrow A_2) \;=\; (A_1 \to A_2) \wedge (A_2 \to A_1)$$

and

$$\neg A \;=\; (A \to \perp).$$

2.3.1 The Hilbert style formulation

We keep the axiom schemes and rule of inference we had for implication and add the axiom schemes

$$\begin{array}{ll}
(Ax3) & A_1 \to A_2 \to (A_1 \wedge A_2) \\
(Ax4) & (A_1 \wedge A_2) \to A_1 \\
(Ax5) & (A_1 \wedge A_2) \to A_2 \\
(Ax6) & A_1 \to (A_1 \vee A_2) \\
(Ax7) & A_2 \to (A_1 \vee A_2) \\
(Ax8) & (A_1 \to C) \to (A_2 \to C) \to (A_1 \vee A_2) \to C \\
(Ax9) & \perp \to A
\end{array}$$

2.3.2 The Natural Deduction formulation

We first give the schemes for the new forms of ND tree-proofs.

$$(\wedge I)\ \frac{A_1 \quad A_2}{A_1 \wedge A_2} \qquad (\wedge E1)\ \frac{A_1 \wedge A_2}{A_1} \qquad (\wedge E2)\ \frac{A_1 \wedge A_2}{A_2}$$

$$(\vee I1)\ \frac{A_1}{A_1 \vee A_2} \qquad (\vee I2)\ \frac{A_2}{A_1 \vee A_2} \qquad (\vee E)\ \frac{A_1 \vee A_2 \quad \begin{matrix}[A_1] \\ C\end{matrix} \quad \begin{matrix}[A_2] \\ C\end{matrix}}{C}$$

$$(\perp E)\ \frac{\perp}{C}$$

Note that $(\vee E)$ involves the discharge of assumption occurrences in the ND tree-proofs of the second and third premisses.

The notion of a **normal** ND tree-proof extends to the full intuitionistic propositional logic after we add the new forms of **redex** for $\wedge$ and $\vee$:

$$\begin{array}{c}\dfrac{\dfrac{\begin{matrix}\vdots\\ A_1\end{matrix}\qquad\begin{matrix}\vdots\\ A_2\end{matrix}}{A_1\wedge A_2}}{A_i}\\ \vdots\end{array}\qquad\qquad\begin{array}{c}\dfrac{\dfrac{\begin{matrix}\vdots\\ A_i\end{matrix}}{A_1\vee A_2}\qquad\begin{matrix}[A_1]\\ \vdots\\ C\end{matrix}\qquad\begin{matrix}[A_2]\\ \vdots\\ C\end{matrix}}{C}\\ \vdots\end{array}$$

where $i = 1, 2$. Their **contracta** are

$$\begin{matrix}\vdots\\ A_i\\ \vdots\end{matrix}\qquad\text{and}\qquad\begin{matrix}\vdots\\ A_i\\ \vdots\\ C\\ \vdots\end{matrix}$$

respectively. With these definitions theorem 3 of section 2 and its methods of proof carry over to full intuitionistic propositional logic.

2.3.3 The sequent formulation of Natural Deduction

We give the new rules of inference in the sequent formulation.

$$(\wedge I)\ \dfrac{\Gamma \mapsto A_1 \qquad \Gamma \mapsto A_2}{\Gamma \mapsto A_1 \wedge A_2}$$

$$(\wedge E1)\ \dfrac{\Gamma \mapsto A_1 \wedge A_2}{\Gamma \mapsto A_1} \qquad (\wedge E2)\ \dfrac{\Gamma \mapsto A_1 \wedge A_2}{\Gamma \mapsto A_2}$$

$$(\vee I1)\ \dfrac{\Gamma \mapsto A_1}{\Gamma \mapsto A_1 \vee A_2} \qquad (\vee I2)\ \dfrac{\Gamma \mapsto A_2}{\Gamma \mapsto A_1 \vee A_2}$$

$$(\vee E)\ \dfrac{\Gamma \mapsto A_1 \vee A_2 \qquad \Gamma, A_1 \mapsto C \qquad \Gamma, A_2 \mapsto C}{\Gamma \mapsto C}$$

$$(\perp E)\ \dfrac{\Gamma \mapsto \perp}{\Gamma \mapsto C}$$

2.3.4 Sequent Calculus Formulation

$$(\wedge R)\ \frac{\Gamma \ \mapsto A_1 \qquad \Gamma \ \mapsto A_2}{\Gamma \ \mapsto A_1 \wedge A_2}$$

$$(\wedge L1)\ \frac{\Gamma, A_1 \ \mapsto C}{\Gamma, (A_1 \wedge A_2) \ \mapsto C} \qquad (\wedge L2)\ \frac{\Gamma, A_2 \ \mapsto C}{\Gamma, (A_1 \wedge A_2) \ \mapsto C}$$

$$(\vee R1)\ \frac{\Gamma \ \mapsto A_1}{\Gamma \ \mapsto A_1 \vee A_2} \qquad (\vee R2)\ \frac{\Gamma \ \mapsto A_2}{\Gamma \ \mapsto A_1 \vee A_2}$$

$$(\vee L)\ \frac{\Gamma, A_1 \ \mapsto C \qquad \Gamma, A_2 \ \mapsto C}{\Gamma, (A_1 \vee A_2) \ \mapsto C}$$

$$(\perp L)\ \ \Gamma, \perp \ \mapsto C$$

Exercise 2.10 *Extend your proofs of theorems 1,2,4,5 of section 2 to the full intuitionistic propositional logic.*

3 Untyped Lambda Calculus

3.1 Preliminaries

3.1.1 The notion of a function

Let A, B be sets. As usual we write $f : A \to B$ if f **is a function from** A **to** B. So we have the rule of **function application**:

$$\frac{f : A \to B \qquad a \in A}{f(a) \in B}$$

Functions are treated **extensionally** so that functions $f, g : A \to B$ are equal if they have the same value on every argument; i.e. if $f, g : A \to B$ then

$$f = g \iff (\forall x \in A)\ f(x) = g(x).$$

A function $f : A \to B$ can be defined by a definition of the form

$$f(x) = \cdots x \cdots \text{ for } x \in A.$$

where '$\cdots x \cdots$' is an expression involving the variable x that denotes a value in the set B whenever x is assigned a value in the set A. It is convenient to write

$$(\lambda x \in A) \cdots x \cdots$$

for the unique function $f : A \to B$ with the above defining equation. So we have the rule of function abstraction:

$$\frac{\begin{array}{c}[x \in A] \\ \cdots x \cdots \in B\end{array}}{(\lambda x \in A) \cdots x \cdots \ :\ A \to B}$$

3.1.2 Examples of functions

Let A, B, C be sets.

Identity Let $I_A = (\lambda x \in A)x$. Then $I_A : A \to A$ is the unique function defined by

$$I_A(x) = x \text{ for } x \in A.$$

Constant Let $K_{A,b} = (\lambda x \in A)b$, where $b \in B$. Then $K_{A,b} : A \to B$ is the unique function defined by

$$K_{A,b}(x) = b \text{ for } x \in A.$$

Composition Let $(g \circ f) = (\lambda x \in A)(g(f(x)))$ where $f : A \to B$ and $g : B \to C$. Then $(g \circ f) : A \to C$ is the unique function defined by

$$(g \circ f)(x) = g(f(x)) \text{ for } x \in A.$$

3.1.3 Functions as sets

Let A, B be sets. The **cartesian product** $A \times B$ is the set of all ordered pairs (a, b) with $a \in A$ and $b \in B$. So we have the rule

$$\frac{a \in A \quad b \in B}{(a, b) \in A \times B}$$

If $f : A \to B$ then its **graph** is the set G of all pairs $(a, f(a))$ with $a \in A$. The set G is a subset of $A \times B$ such that

$$(\forall x \in A)(\exists! y \in B) \;\; (x, y) \in G.$$

Conversely for any subset G of $A \times B$ that satisfies the above condition we can define a function $f : A \to B$ with the defining equation

$$f(x) = \text{ the unique } y \in B \text{ such that } (x, y) \in G.$$

In axiomatic set theory functions are identified with their graphs.

3.1.4 Multi-argument functions

Let $A_1, \ldots, A_n, B$ be sets. The cartesian product $A_1 \times \cdots \times A_n$ is the set of all ordered n-tuples $(a_1, \ldots, a_n)$ with $a_1 \in A_1, \ldots, a_n \in A_n$. If $f : A_1 \times \cdots \times A_n \to B$ then we have the rule

$$\frac{a_1 \in A_1 \;\; \cdots a_n \in A_n}{f(a_1, \ldots, a_n) \in B}$$

If $\cdots$ is an expression for a value in B when variables $x_1, \ldots, x_n$ that may occur in $\cdots$ are assigned values in $A_1, \ldots, A_n$ respectively then

$$(\lambda(x_1, \ldots, x_n) \in A_1 \times \cdots \times A_n) \;\; \cdots$$

is the unique function $f : A_1 \times \cdots \times A_n \to B$ with the defining equation

$$f(x_1, \ldots, x_n) = \;\; \cdots.$$

3.1.5 Currying

Let A, B, C be sets. We write C^B for the set of all the functions $B \to C$. If $f : A \times B \to C$ then $g : A \to C^B$ where

$$g = (\lambda x \in A)(\lambda y \in B) f(x, y).$$

We call g the **curried** version of f. This term is named after the American logician Haskell B Curry, who was the main developer of *Combinatory Logic*. His name has also been used for the functional programming language *Haskell*. The function f can be recaptured from g by a process called **uncurrying**:

$$f = (\lambda(x, y) \in A \times B) g(x)(y)$$

3.1.6 The problem of variable clashes

Let A be a set and let $f : A \times A \to A$. The curried version $g : A \to A^A$ has defining equation

$$(1) \quad g(x) = (\lambda y \in A) f(x, y) \text{ for } x \in A.$$

We could use 'z' instead of 'y':-

$$(2) \quad g(x) = (\lambda z \in A) f(x, z) \text{ for } x \in A.$$

But we cannot use 'x' instead of 'y':-

$$(3) \quad g'(x) = (\lambda x \in A) f(x, x) \text{ for } x \in A.$$

The function g' defined in (3) is generally quite different to the function g defined in (1) or in (2). In (3) the variable 'x' occurs free on the left hand side while it only occurs bound in the lambda expression on the right hand side so that g' is a constant function.

If $h : A \to A$ let $k : A \to A$ be given by

$$k(y) = g(h(y)) \text{ for } y \in A.$$

Using (1) to expand the right hand side we get:-

$$k(y) = (\lambda y \in A) f(h(y), y) \text{ for } y \in A,$$

which is wrong! Instead we can use (2) to get

$$k(y) = (\lambda z \in A) f(h(y), z) \text{ for } y \in A.$$

Moral: Before making a textual substitution you must first suitably relabel bound variables so as to avoid variable clashes. This same problem of variable clashes is a familiar feature of quantifiers in the predicate calculus.

3.2 An Untyped Universe

The (untyped) **Lambda Calculus** (LC) is a calculus of 'functions' where it 'makes sense' to 'apply' anything to anything. To motivate LC we postulate a non-trivial set U (so U should have more than one element) such that $U = U^U$. This will be our universe of functions. For $a, b \in U$ we write $a \cdot b$ or usually just ab for the result of applying the function $a : U \to U$ to $b \in U$. We use the following abbreviations:-

$ab_1 b_2 \cdots b_n$	for	$\cdots((ab_1)b_2)\cdots b_n$
$\lambda x.\ \cdots$	for	$(\lambda x \in U)\ \cdots$
$\lambda x_1 \cdots x_n.\ \cdots$	for	$\lambda x_1. \cdots \lambda x_n.\ \cdots$

Note that if $f : U^n \to U$ and $a = \lambda x_1 \cdots x_n . f(x_1, \ldots, x_n)$ then $a \in U$ and, for $x_1, \ldots, x_n \in U$

$$f(x_1, \ldots, x_n) = a x_1 \cdots x_n.$$

So $f = (\lambda(x_1, \ldots, x_n) \in U^n) a x_1 \cdots x_n$. We call a the **curried** version of f.

3.2.1 Some combinators in U

$$\begin{array}{lll} I = \lambda x.x, & Ia & = a \\ K = \lambda xy.x, & Kab & = a \\ B = \lambda xyz.x(yz), & Babc & = a(bc) \\ C = \lambda xyz.xzy, & Cabc & = acb \\ S = \lambda xyz.xz(yz), & Sabc & = ac(bc) \end{array}$$

The above λ-expressions are examples of **combinators**. Following their definitions are their defining equations, where a, b, c are arbitrary elements of U. There can be equations connecting the combinators. For example we have the following two results.

Proposition 3.1 $I = SKK$.

Proof: Observe that for any $x \in U$

$$SKKx = Kx(Kx) = x.$$

But I is the unique element of U such that $Ix = x$ for all $x \in U$. It follows that $SKK = I$.

Proposition 3.2 $C = S(BBS)(KK)$.

Proof: Observe that for any $x, y, z \in U$

$$\begin{aligned} S(BBS)(KK)xyz &= BBSx(KKx)yz \\ &= B(Sx)(KKx)yz \\ &= B(Sx)Kyz \\ &= Sx(Ky)z \\ &= xz(Kyz) \\ &= xzy \\ &= xyz. \end{aligned}$$

So we get the result.

Combinatory Logic (CL) is concerned with the combinators such as I, K, B, C, S and equations involving them.

3.2.2 Some laws of LC

$$(\beta) \qquad (\lambda x. \cdots x \cdots)a = \cdots a \cdots$$

$$(\eta) \qquad \lambda x.(ax) = a$$

$$(ext) \qquad \frac{\begin{array}{c}[x \in A] \\ ax = bx\end{array}}{a = b}$$

Exercise 3.3 *Let $V = CB(SII)$. Given $a \in U$ let $d = Va$ and let $c = dd$. Show that $c = ac$. Show that $c = Ya$.*

We call c a **fixed point** of a and we call Y a **fixed point combinator**.

3.2.3 Problems with "$U = U^U$"

In the standard axiomatic set theory ZFC it is inconsistent to have a non-trivial set U such that $U = U^U$. But it is probably relatively consistent in the variant of ZFC that uses intuitionistic logic instead of clasical logic and drops the Foundation Axiom. We give two proofs of the inconsistency.

First Contradiction in ZFC Suppose that $a \in U \subseteq U^U$ and let $b = a(a)$. Then $(a, b) \in a$. So, as $(a, b) = \{\{a\}, \{a, b\}\}$,

$$a \in \{a\} \in (a, b) \in a$$

contradicting the Foundation Axiom.

Second Contradiction in ZFC This proof uses cardinal numbers. Let $n > 1$ be the cardinality of U. If $U = U^U$ then $n = n^n$. But by Cantor's theorem $2^n > n$ so that $n^n \geq 2^n > n$, contradicting $n = n^n$.

Note that the reasoning in the first proof is constructive. But a closer examination of the cardinality argument in the second proof will show that the second proof is not constructive.

3.2.4 A more general universe

This time we assume that U is a non-trivial set with functions $F : U \to U^U$ and $G : U^U \to U$ such that $F \circ G$ is the identity function on U^U. We can define

$$\begin{array}{lll} ab & = F(a)(b) & \text{for } a, b \in U \\ \lambda x.f(x) & = G(f) & \text{for } f \in U^U \end{array}$$

We still have

$$(\beta) \quad (\lambda x.f(x))a = f(a).$$

But only have weak versions of (η) and (ext). Let

$$U' = \{G(f) \mid f \in U^U\}.$$

Then we have

$(w\eta)$ If $a \in U'$ then $a = \lambda x.(ax)$.

$(wext)$ If $a, b \in U'$ and $ax = bx$ for all $x \in U$ then $a = b$.

When $U' = U$ then we do get (η) and (ext) and F is a bijection $F : U \cong U^U$, with inverse G.

3.3 Syntax of LC

3.3.1 Terms

We assume given a set const of constants and an infinite set var of variables. The **terms** of LC are built up from the constants and variables using application and lambda abstraction. So the terms M are given by the BNF style grammar

$$M ::= \ x \mid c \mid (MM) \mid (\lambda x.M)$$

where c is used for constants and x for variables. Alternatively the set of terms is inductiveley defined using the following rules.

1. Every constant is a term and every variable is a term.
2. If M_1, M_2 are terms then so is $(M_1 M_2)$.
3. If x is a variable and M is a term then $(\lambda x.M)$ is a term.

Notational Conventions

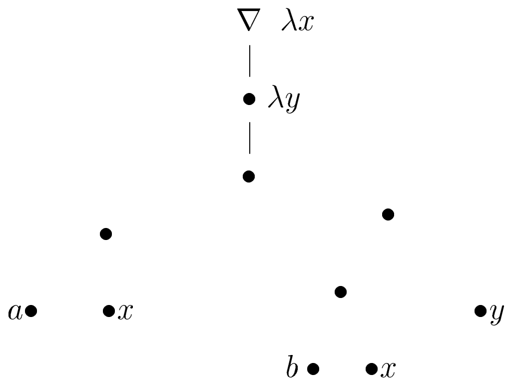
- Leave off outermost parentheses from terms.
- $MM_1M_2 \cdots M_n$ abbreviates $((\cdots((MM_1)M_2)\cdots)M_n)$
- $\lambda x_1 \cdots x_n.M$ abbreviates $(\lambda x_1.(\cdots.(\lambda x_n.M)\cdots))$.

Example: $\lambda xy.(ax(bxy))$ abbreviates

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

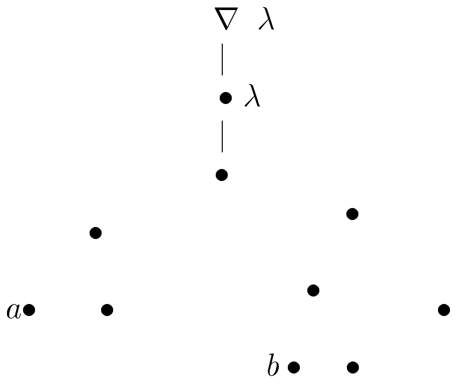
3.3.2 Construction Trees

Each term is constructed in a unique way using the rules and so has an associated **construction tree**. For example the term $(\lambda x.(\lambda y.((ax)((bx)y)))$ has the tree



Note that ∇ indicates explicitly the root of the downward growing tree. Each leaf is labelled with a variable or constant.

A variable leaf $\bullet y$ is **bound** if there is a λ-node $\bullet\lambda y$ in the path from the leaf to the root of the tree. Otherwise the leaf is free. If a leaf $\bullet y$ is bound then the first λ-node $\bullet\lambda y$, in the path from the leaf to the root, is called the **binder** of the leaf.
The only formal purpose of bound variables is to specify the binders of bound leaf nodes. So, instead of using construction trees with bound variables we may use **construction graphs** with a pointer up from each bound leaf nodes to their binder. The occurrences of the bound variables can then be removed. So, for example, the previous tree has the following graph.



This is the construction graph of $(\lambda x.(\lambda y.((ax)((bx)y))))$.

As another example here is the construction graph of $\lambda y.(xy(\lambda y.y))$.

The terms $\lambda z.(xz(\lambda x.x))$ and $\lambda v.(xv(\lambda x.x))$ have the same construction graph.

Such terms are said to be **α-convertible**. Here is yet another example of a construction graph and a pair of α-convertible terms for it.

1. $\lambda x.(xz(\lambda y.(xy)))$
2. $\lambda w.(wz(\lambda x.(wx)))$

The term 2 can be obtained from the term 1 by simultaneously relabelling x as w and y as x. Next we give three alternative notations for the graph.

Bourbaki $\lambda(\Box z(\lambda(\Box\Box)))$

de Bruijn $\lambda(1z(\lambda(21)))$

dual de Bruijn $\lambda(1z(\lambda(12)))$

The **de Bruijn index** of a bound leaf in a construction tree is the number of λ-nodes occuring in the path from the leaf up to and including the binder of the leaf node. The **dual de Bruijn index** is instead the number of λ-nodes in the path from and including the binder up to the root.

The **(dual) de Bruijn term** associated with a term is obtained by replacing each variable occuring at a bound leaf by the (dual) de Bruijn index of the leaf and then erasing the variable from each binder.

3.3.3 The de Bruijn terms

We first define the de Bruijn **pre-terms** using the BNF-style grammar equation

$$M ::= c \mid x \mid (MM) \mid (\lambda M) \mid 1 \mid 2 \mid 3 \mid \cdots .$$

For each variable x and each pre-term M let $(\lambda x.M)$ be the pre-term obtained from (λM) by simultaneously making the replacements

$$\begin{array}{c} x \to 1 \\ 1 \to 2 \\ 2 \to 3 \\ \vdots \end{array}$$

Example: If M is $\lambda(1z(\lambda(12)))$ then $\lambda z.M$ is $\lambda(\lambda(21(\lambda(23))))$.

Note that we are here preferring to take the dual de Bruijn approach. The set of **de Bruijn terms** is now defined to be the set of pre-terms inductively defined using the same rules as we used earlier to define the terms of LC, except now the lambda abstraction rule has to be understood in the sense we have just explained. So now there is no such thing as a bound variable in a de Bruijn term, only perhaps in our notation to refer to it. Instead the numerical indices are used.

From now on we will use the de Bruijn terms. For each pre-term M let $\mathsf{var}(M)$ be the set of variables that occur in M. Also let $M\{y/x\}$ be the result of replacing all occurrences of x in M by y.

Proposition 3.4 *$\lambda x.M = \lambda y.(M\{y/x\})$ if $y \notin \mathsf{var}(M)$.*

Proposition 3.5 *If (λN) is a (de Bruijn) term and $x \notin \mathsf{var}(N)$ then there is a unique term M such that $(\lambda N) = (\lambda x.M)$.*

3.3.4 Substitution

Given a variable y and a term N the function $M \mapsto M[N/y]$ from terms to terms is the unique function $(\ )'$ such that

$$\begin{cases} c' & = c \\ y' & = N \\ x' & = x & \text{if } x \neq y \\ (M_1 M_2)' & = M_1' M_2' \\ (\lambda x.M)' & = (\lambda x.M') & \text{if } x \notin \mathsf{var}(N) \cup \{y\} \end{cases}$$

Proposition 3.6 *The function $M \mapsto M[N/y]$ is well defined.*

Note that this is not literal substitution as indices in N get shifted up 1 for each λ above it in the construction tree of N.

Lemma 3.7 (The Substitution Lemma)

$$M[N/x][L/y] = M[L/y][N[L/y]/x]$$

if $x \neq y$ and $x \notin \mathsf{var}(L)$.

Proof: By induction on the structure of M.

3.4 Deduction

The $\lambda\beta$-**calculus** is the formal system, whose statements are equations between terms, having the following axiom schemes and rules of inference.

(1) $M = M$

(2) $\dfrac{M = N}{N = M}$

(3) $\dfrac{M = N \quad N = L}{M = L}$

(4) $\dfrac{M_1 = N_2 \quad M_2 = N_2}{M_1 M_2 = N_1 N_2}$

(ξ) $\dfrac{M = N}{\lambda x.M = \lambda x.N}$

(β) $(\lambda x.M)N = M[N/x]$

The **$w\lambda\beta$-calculus** is just like the $\lambda\beta$-calculus except that the rule (ξ) is left out. The **$\lambda\beta\eta$-calculus** is obtained from the $\lambda\beta$-calculus by adding the axiom scheme

(η) $\lambda x.(Mx) = M$, provided that $x \notin \mathsf{var}(M)$.

We will also consider the following rule.

$(ext)\ \dfrac{Mx = Nx}{M = N}$, provided that $x \notin \mathsf{var}(M) \cup \mathsf{var}(N)$.

There are weak versions $(w\eta)$ and $(wext)$ of (η) and (ext), where it is required that M, N have λ-form; i.e. are terms of the form $\lambda x.L$.

The Consistency Problem: Show that there are terms M, N such that $M = N$ is not a theorem of $(\lambda\beta\eta)$.

Exercises 3.8

1. *Show that* $(\lambda\beta\eta) \equiv (w\lambda\beta)+(ext)$*; i.e. both sides have the same theorems.*
2. *Show that*
   (a) $(\lambda\beta) \vdash (w\eta)$,
   (b) $(\lambda\beta) \equiv (w\lambda\beta) + (wext)$.
3. *If* $Y = \lambda z.((\lambda x.z(xx))(\lambda x.z(xx)))$ *show that for any term* M
$$(w\lambda\beta) \vdash M(YM) = YM.$$
4. *If* S, K, I *are the terms given in section 2 show that*
$$(\lambda\beta) \vdash I = SKK.$$
   Can you show that
$$(w\lambda\beta) \vdash I = SKK?$$

3.5 Combinatory Logic

Very roughly, combinatory logic is an approach to the ideas of the lambda calculus that avoids lambda abstraction and bound variables. This means that substitution can be understood literally. The syntax of combinatory logic is given by the grammar
$$M ::= \ c \mid K \mid S \mid x \mid (MM)$$
So, instead of lambda abstraction there are the special symbols K, S. It turns out that in the formal system CL for combinatory logic, defined below, we can

define a simulation of lambda abstraction so that $(w\lambda\beta)$ can be translated into CL. The main axioms for CL are the defining equations for K and S. There is an easy translation of CL into $(w\lambda\beta)$, so that these two formal systems are closely related. But the two translations are not exactly inverses of each other. Really $(w\lambda\beta)$ is a **conservative extension** of a subsystem that is equivalent to CL. The rule (ext) also makes sense for combinatory logic and we can show that $CL + (ext)$ is equivalent to $(\lambda\beta\eta)$. Also we can define a combinatory logic weakening $(wext)$ of (ext) and show that $CL + (wext)$ is equivalent to $(\lambda\beta)$.

3.5.1 CL and its translation into $(w\lambda\beta)$

The formal system CL consists of the axiom scheme and rules **(1)-(4)** from section 4, for equations between terms of combinatory logic, together with the axiom schemes

$$\begin{array}{lll} (K) & KMN & = M \\ (S) & SMNL & = ML(NL) \end{array}$$

where M, N, L are arbitrary combinatory logic terms. Below we will call such terms CL**-terms** in contrast to the terms of the lambda calculus which we will call LC**-terms**.

The translation $M \mapsto M_{LC}$ from CL-terms to LC terms is defined by structural recursion using the equations

$$\begin{cases} c_{LC} & = c \\ K_{LC} & = \lambda xy.x \\ S_{LC} & = \lambda xyz.xz(yz) \\ x_{LC} & = x \\ (MN)_{LC} & = M_{LC}N_{LC} \end{cases}$$

It is straightforward to prove the following result.

Proposition 3.9 $CL \vdash M = N \implies (w\lambda\beta) \vdash M_{LC} = N_{LC}$.

3.5.2 A translation of $(w\lambda\beta)$ into CL

In order to give such a translation we need to simulate lambda abstraction in CL. For each variable x we define the operation $M \mapsto \lambda^*x.M$ on CL-terms by structural recursion using the following equations.

$$\begin{cases} \lambda^*x.x & = I & \text{(where } I \text{ is } SKK) \\ \lambda^*x.M & = KM & (x \notin \mathsf{var}(M)) \\ \lambda^*x.(MN) & = S(\lambda^*x.M)(\lambda^*x.N) & (x \in \mathsf{var}(MN)) \end{cases}$$

Lemma 3.10 *For all CL-terms M, N*

$$CL \vdash (\lambda^*x.M)N = M[N/x].$$

The translation $M \mapsto M_{CL}$ of LC-terms into CL-terms is by structural recursion using the equations

$$\begin{cases} c_{CL} & = c \\ x_{CL} & = x \\ (MN)_{CL} & = M_{CL}N_{CL} \\ (\lambda x.M)_{CL} & = \lambda^* x.M_{CL} \end{cases}$$

Proposition 3.11 $(w\lambda\beta) \vdash M = N \implies CL \vdash M_{CL} = N_{CL}$.

3.5.3 The converse of Proposition 6

We have seen translations both ways between CL and $(w\lambda\beta)$. When extensionality is added to both formal systems then the translations carry over and are inverses of each other in a suitable sense, showing that $CL + (ext)$ is equivalent to $(w\lambda\beta) + (ext)$; i.e. $(\lambda\beta\eta)$. But the translations between CL and $(w\lambda\beta)$ are not quite inverses. By using a variant of λ^* we now show that '$\Longrightarrow$' in proposition 6 can be strengthened to '$\Longleftrightarrow$'. In the variant definition of λ^* we use the equation

$$\lambda^* x.(Mx) = M \quad (x \notin \mathsf{var}(M))$$

and only use the third equation in the previous definition of λ^* when the above equation does not apply.

Exercise 3.12 *Show, using this variant definition of λ^* when defining $M \mapsto M_{CL}$, that*

$$\begin{cases} (K_{LC})_{CL} & = K \\ (S_{LC})_{CL} & = S \end{cases}$$

and hence that

$$(M_{LC})_{CL} = M$$

for all CL-terms M. Note that these are syntactic identities and not equalities proved in CL.

Proposition 3.13

$$CL \vdash M = N \iff (w\lambda\beta) \vdash M_{LC} = N_{LC}$$

for all CL-terms M, N.

Proof: Note that Proposition 7 still holds when using the variant definition of λ^*. So if $(w\lambda\beta) \vdash M_{LC} = N_{LC}$ then $CL \vdash (M_{LC})_{CL} = (N_{LC})_{CL}$. So, by the exercise, $CL \vdash M = N$.

Exercise 3.14 *Show that $(w\lambda\beta)$ can be replaced by $(w\lambda\beta) + (\eta)$ in Proposition 7 and hence also in Proposition 8.*

3.5.4 The equivalence between $CL + (ext)$ and $(\lambda\beta\eta)$

Proposition 3.15

1. *$CL + (ext) \vdash M = N \implies (\lambda\beta\eta) \vdash M_{LC} = N_{LC}$.*
2. *$(\lambda\beta\eta) \vdash M = N \implies CL + (ext) \vdash M_{CL} = N_{CL}$.*
3. *$CL + (ext) \vdash M = (M_{LC})_{CL}$ for any CL-term M.*
4. *$(\lambda\beta\eta) \vdash M = (M_{CL})_{LC}$ for any LC-term M.*
5. *'$\implies$' in 1,2 can be replaced by '$\iff$'.*

3.5.5 The equivalence between $CL + (wext)$ and $(\lambda\beta)$

In combinatory logic the rule $(wext)$ is the weakening of the rule (ext) which requires that M, N are functional in the sense of the following definition.

Definition 3.16 *A CL-term is* **functional** *if it has one of the forms*

$$K, KM, S, SM, SMN.$$

Observe that, when using the original definition of λ^*, a CL-term $\lambda^* x.M$ is always functional so that M_{CL} is a functional term for any LC-term M that is in λ-form. But note that this observation does not work with the variant definition of λ^*!

Exercise 3.17 *Show that for each functional CL-term M there is a LC-term N in λ-form such that*

$$(\lambda\beta) \vdash M_{CL} = N.$$

Proposition 3.18 *The results of Proposition 9 hold when (ext) is replaced by $(wext)$ and $(\lambda\beta\eta)$ is replaced by $(\lambda\beta)$.*

3.5.6 Summary

The main relationships between combinatory logics and lambda calculi are summarised in the following diagram.

$$\begin{array}{ccccc}
CL & \subseteq & CL + (wext) & \subseteq & CL + (ext) \\
\Downarrow\uparrow & & \Updownarrow & & \Updownarrow \\
w\lambda\beta & \subseteq & \lambda\beta & \subseteq & \lambda\beta\eta
\end{array}$$

3.6 The Church-Rosser Theorem

The aim of this section is to prove the result of the title for the lambda calculi $(\lambda\beta)$ and $(\lambda\beta\eta)$. This result, in each case, has the immediate consequence that the calculus is consistent. Here, by the **consistency** of an equational calculus we mean the property that not all equations can be proved, or equivalently, that for distinct variables x, y the equation $(x = y)$ cannot be proved.

A **redex** for $(\lambda\beta)$ is a term of the form

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

and its **contractum** is the term $M[N/x]$. These are the **β-redexes**. For $(\lambda\beta\eta)$ there are also the **η-redexes**; i.e. terms of the form

$$\lambda x.(Mx)$$

where $x \notin \mathsf{var}(M)$, having contractum M. For either calculus, given terms M, N we write that M **contracts to** M', abbreviated

$$M \; contr \; M',$$

if M' can be obtained from M by replacing an occurrence of a redex in M by its contractum. Also we write that M **reduces to** M', abbreviated

$$M \; red \; M'$$

if M' can be obtained from M by a sequence of zero, one or more contractions

$$M \; contr \; \cdots \; contr \; M'.$$

So the reduction relation *red* on terms is the reflexive, transitive closure of the contraction relation *contr*. We can now state the theorem.

Theorem 3.19 (Church-Rosser) *For both $(\lambda\beta)$ and $(\lambda\beta\eta)$, if both $M \; red \; N$ and $M \; red \; K$ then there is a term L such that both $N \; red \; L$ and $K \; red \; L$.*

Note: Call a relation R on a set A **confluent on** A if

$$xRy, z \Longrightarrow \exists w{\in}A \; (y, zRw),$$

where xRy, z abbreviates $(xRy \wedge xRz)$ and y, zRw abbreviates $(yRw \wedge zRw)$. Then the above Church-Rosser Theorem states that the reduction relation is confluent for both calculi.

Exercise 3.20 *Given a relation R on a set A let $\sim_R$ be the relation on A given by*

$$y \sim_R z \iff \exists w{\in}A \; (y, zRw).$$

Show that R is confluent iff $\sim_R$ is an equivalence relation. Use the Church-Rosser Theorem to show that for both calculi

$$M \sim_{red} N \iff \vdash M = N.$$

Hence show that each calculus is consistent.

3.7 Normalisation

The definitions in this section apply to each of the calculi $(\lambda\beta)$ and $(\lambda\beta\eta)$. A term is **normal** if no subterm is a redex. If M *red* M' and M' is normal then we call M' a **normal form of** M. A term M is **(weakly) normalisable** if M has a normal form and is **strongly normalisable** if every contraction path

$$M \; contr \; M' \; contr \; M'' \cdots$$

is finite.

Some Examples: Trivially all normal terms are normalisable and strongly normalisable. Examples of normal terms are

$$I = \lambda x.x, \; K = \lambda xy.x, \; S = \lambda xyz.xz(yz), \; \Omega_0 = \lambda x.(xx).$$

Note that $\Omega = \Omega_0\Omega_0$ is not normal. In fact Ω *contr* Ω so that Ω is not normalisable or strongly normalisable. Note that if $K' = KK\Omega$ then K' is not normal, but is normalisable as K' *contr* K. In contrast, as K' *contr*K', the term K' is not strongly normalisable.

Exercises 3.21

1. *Show that a term is normal in the* $(\lambda\beta)$ *calculus iff it has the form*

   $$\lambda x_1 \cdots x_n.aN_1 \cdots N_k$$

   where $n, k \geq 0$, $x_1, \ldots, x_n$ *are variables,* a *is a constant or variable and* $N_1, \ldots, N_k$ *are normal terms. What happens in the* $(\lambda\beta\eta)$ *calculus?*

2. *Show that every normalisable term has a unique normal form.*

3. *Show that every strongly normalisable term is normalisable.*

4. *Show that if* M *is strongly normalisable then the set of its contraction paths*

   $$M \; contr \; M' \; contr \; M'' \cdots$$

   is finite. [Hint: Use König's Lemma]

5. *Show that, for each of the two calculi, it is decidable whether or not an equation* $(M = N)$ *between normalisable terms* M, N*, is provable.*

4 Simply Typed Lambda Calculus

4.1 The Simple Type Theory *STT*

We assume given a set of **atomic types** and generate the types from the atomic types using the rule

If A_1, A_2 are types then so is $(A_1 \to A_2)$

The **typing judgments** have the form $M : A$ where M is a term of the untyped lambda calculus and A is a type. In particular a **variable declaration** is a typing judgment of the form $x : A$ where x is a variable. A sequence

$$x_1 : A_1, \ldots, x_n : A_n$$

of zero, one or more variable declarations, for a non-repeating list $x_1, \ldots, x_n$ of variables, is called a **context**.

The type theory *STT* is a formal system whose statements have the form

$$\Gamma \;\vdash\!\!\blacktriangleright M : A$$

where Γ is a context and $M : A$ is a typing judgment. The axioms and rules of *STT* are given by the following schemes.

$$(ass) \qquad \Gamma \;\vdash\!\!\blacktriangleright x : A \qquad\qquad (x : A \text{ in } \Gamma)$$

$$(appl) \qquad \frac{\Gamma \;\vdash\!\!\blacktriangleright M : (B \to C) \qquad \Gamma \;\vdash\!\!\blacktriangleright N : B}{\Gamma \;\vdash\!\!\blacktriangleright (MN) : C}$$

$$(abstr) \qquad \frac{\Gamma, y : B \;\vdash\!\!\blacktriangleright M : C}{\Gamma \;\vdash\!\!\blacktriangleright \lambda y.M : (B \to C)}$$

Exercises 4.1

1. *In this exercise identify the (atomic) formulae of Intuitionistic Implicational Logic with the (atomic) types of STT. Let $x_1, x_2, \ldots$ be an infinite non-repeating list of variables. Show that*

$$A_1, \ldots, A_n \;\vdash\!\!\blacktriangleright_{ND} A \iff \exists M[x_1 : A_1, \ldots, x_n : A_n \;\vdash\!\!\blacktriangleright_{STT} M : A].$$

   Moreover show that for all sequents $A_1, \ldots A_n \;\vdash\!\!\blacktriangleright A$ there is a one-one correspondence between the ND tree-proofs of $A_1, \ldots, A_n \;\vdash\!\!\blacktriangleright A$ and pairs consisting of a term M and an STT tree-proof of $x_1 : A_1, \ldots, x_n : A_n \;\vdash\!\!\blacktriangleright M : A$.

2. *In the following let Γ be a context $x_1 : A_1, \ldots, x_n : A_n$. Prove the following results.*

(a) If $\Gamma \mapsto_{STT} M : A$ *then* $\mathsf{var}(M) \subseteq \{x_1, \dots, x_n\}$.

(b) If Γ' *is a context that includes every variable declaration of* Γ *then*

$$\Gamma \mapsto_{STT} M : A \implies \Gamma' \mapsto_{STT} M : A.$$

(c) If $\Delta \mapsto_{STT} N_i : A_i$ *for* $i = 1, \dots, n$ *then*

$$\Gamma \mapsto_{STT} M : A \implies \Delta \mapsto_{STT} M[N_1, \dots N_n/x_1, \dots x_n] : A.$$

(d) If M *red* M' *then*

$$\Gamma \mapsto STTM : A \implies \Gamma \mapsto_{STT} M' : A.$$

4.2 The variant STT'

The terms of STT' are like the untyped terms except that lambda abstractions have to include a typing of the variable being bound and so have the form $\lambda x : A.M$, where A is a type. The rules of STT' are just as the rules of STT except that the rule $(abstr)$ has to be replaced by the following rule.

$$(abstr)' \quad \frac{\Gamma, y : B \mapsto M : C}{\Gamma \mapsto \lambda y : B.M : (B \to C)}$$

Exercises 4.2

1. *If* M *is a term of* STT' *then let* M^- *be the untyped term obtained by erasing '*$: A$*' from each abstraction subterm* $\lambda x : A.N$. *Show that*

$$\Gamma \mapsto_{STT} M : A \iff \exists M'[M = M'^- \text{ and } \Gamma \mapsto_{STT'} M' : A].$$

2. *Show that if there is a type* A *such that* $\Gamma \mapsto_{STT'} M : A$ *then there is a unique such* A *and for that* A *there is a unique tree-proof in* STT' *of* $\Gamma \mapsto M : A$.

By the result of the second exercise we may write $type_\Gamma(M)$ for the unique type A such that $\Gamma \mapsto_{STT'} M : A$, when it exists. Also note that if Γ is the context $x_1 : A_1, \dots, x_n : A_n$ then the terms M of STT' such that $\Gamma \mapsto_{STT'} M : A$ are in one-one correspondence with the ND tree-proofs of $A_1, \dots, A_n \mapsto A$. So these terms M can be viewed as formal expressions that represent the corresponding ND tree-proofs.

4.2.1 Standard Set Theoretical Models of STT'

Given an assignment of a set $[[A_0]]$ to each atomic type A_0 we may define a **standard set theoretical model for** STT' in the following way. Define $[[A]]$ for each type A by structural recursion using the equation

$$[[A_1 \to A_2]] = [[A_2]]^{[[A_1]]},$$

where, for sets X, Y, Y^X is the set of all the functions from X to Y. Also, if Γ is the context $x_1 : A_1, \ldots, x_n : A_n$ then let

$$[[\Gamma]] = [[A_1]] \times \cdots \times [[A_n]].$$

Now, by structural recursion on M, whenever $\Gamma \;\mapsto\!\!\blacktriangleright_{STT'} M : A$ we may assign a function

$$[[M]]_\Gamma : [[\Gamma]] \to [[A]]$$

using the following equations, where we let $\vec{a} = (a_1, \ldots, a_n) \in [[\Gamma]]$.

$$\begin{cases} [[x_i]]_\Gamma(\vec{a}) & = a_i \qquad (i = 1, \ldots, n) \\ [[(MN)]]_\Gamma(\vec{a}) & = [[M]]_\Gamma(\vec{a})([[N]]_\Gamma(\vec{a})) \\ [[\lambda y : B.M]]_\Gamma(\vec{a}) & = (\boldsymbol{\lambda} b \in [[B]])[[M]]_{\Gamma, y:B}(\vec{a}, b) \end{cases}$$

where, in the last equation the right hand side is the function $f : [[B]] \to [[C]]$ such that for all $b \in [[B]]$

$$f(b) = [[M]]_{\Gamma, y:B}(\vec{a}, b).$$

Note that it would not be so easy to formulate a notion of set theoretical model for the type theory STT.

4.3 Standard Term Models of STT

Call an untyped term M an STT-term if $\Gamma \;\mapsto\!\!\blacktriangleright_{STT} M : A$ for some context Γ and some type A. In this section we introduce a notion of term model for STT that can be used to prove such results as the following theorem.

Theorem 4.3 (Normalisation Theorem for STT**)** *Every STT-term is normalisable.*

Let $\mathcal{T}$ be the set of untyped terms. For subsets X, Y of $\mathcal{T}$ let $X \to Y$ and $X \to^w Y$ be given by the following definitions.

$$\begin{array}{ll} X \to Y & = \{M \mid \forall N \in X \; (MN) \in Y\} \\ X \to^w Y & = \{\lambda x.M \mid \forall N \in X \; M[N/x] \in Y\} \end{array}$$

Lemma 4.4 *If $X' \subseteq X \subseteq \mathcal{T}$ and $Y \subseteq Y' \subseteq \mathcal{T}$ then $(X \to Y) \subseteq (X' \to Y')$.*

Proof: Let $X' \subseteq X \subseteq \mathcal{T}$ and $Y \subseteq Y' \subseteq \mathcal{T}$ and let $M \in (X \to Y)$. So $MN \in Y$ for each $N \in X$. Now let $N \in X'$. Then $N \in X$ so that $MN \in Y$ and hence $MN \in Y'$. Thus $M \in (X' \to Y')$.

■

Definition 4.5 *A* **standard term model of** *STT is an assignment of a subset $[[A]]$ of $\mathcal{T}$ to each type A such that for all types A_1, A_2*

$$[[A_1]] \to^w [[A_2]] \ \subseteq \ [[A_1 \to A_2]] \ \subseteq \ [[A_1]] \to [[A_2]].$$

Given such a model let $x_1 : A_1, \ldots x_n : A_n \models M : A$ if, for all $N_1 \in [[A_1]], \ldots, N_n \in [[A_n]]$

$$M[N_1, \ldots, N_n/x_1, \ldots, x_n] \in [[A]].$$

Theorem 4.6 (Soundness) *Given a standard term model of STT*

$$\Gamma \ \mapsto_{STT} M : A \implies \Gamma \models M : A.$$

Proof: This is by a straightforward induction on the size of an STT tree-proof of $\Gamma \ \mapsto M : A$. The base case, for the axioms (ass), is trivial. The induction step for the rule $(appl)$ uses

$$[[B \to C]] \ \subseteq \ [[B]] \to [[C]].$$

The induction step for the rule $(abstr)$ uses

$$[[B]] \to^w [[C]] \ \subseteq \ [[B \to C]].$$

But some care is needed to relabel the variable y, given an instance of the rule

$$\frac{\Gamma, y : B \ \mapsto M : C}{\Gamma \ \mapsto \lambda y.M : (B \to C)}.$$

The induction hypothesis is $\Gamma, y : B \models M : C$. If Γ is the context $x_1 : A_1, \ldots, x_n : A_n$ then this means that for all $N_1 \in [[A_1]], \ldots, N_n \in [[A_n]]$

$$(*) \qquad (\forall N \in [[B]]) \ M[N_1, \ldots, N_n, N/x_1, \ldots, x_n, y] \in [[C]].$$

We can rewrite $(*)$ as

$$(\forall N \in [[B]]) M'[N/y'] \in [[C]]$$

where

$$M' = M[y'/y][N_1, \ldots, N_n/x_1, \ldots, x_n],$$

provided that y' is a fresh variable (i.e. distinct from each of $x_1, \ldots, x_n, y$ and not occurring in any of $M, N_1, \ldots, N_n, N$). So we get from (*) that

$$\lambda y'.M' \in [[B]] \to^w [[C]] \subseteq [[B \to C]].$$

Now observe that

$$\lambda y'.M' = (\lambda y.M)[N_1, \ldots, N_n/x_1, \ldots, x_n].$$

It follows that for all $N_1 \in [[A_1]], \ldots, N_n \in [[A_n]]$

$$(\lambda y.M)[N_1, \ldots, N_n/x_1, \ldots, x_n]) \in [[B \to C]]$$

so that we have $\Gamma \models \lambda y.M : (B \to C)$ as required.

■

Definition 4.7 *A collection $\mathcal{C}$ of sets of terms is an STT-***collection** *if the following conditions hold.*

1. *There is at least one set in $\mathcal{C}$.*
2. *$Var \subseteq X$ for all $X \in \mathcal{C}$.*
3. *There is a binary operation m on $\mathcal{C}$ such that for all $X, Y \in \mathcal{C}$*

$$X \to^w Y \;\subseteq\; m(X, Y) \;\subseteq\; X \to Y.$$

Theorem 4.8 *Let S be a set of terms such that there is an STT-collection of subsets of S. Then every STT-term is in S.*

Proof: Let $\mathcal{C}$ be an STT-collection of subsets of S and let $X_0 \in \mathcal{C}$. We may define a standard term model of STT as follows. We recursively define $[[A]]$ for all types A by letting $[[A_0]] = X_0$ for each atomic type A_0 and letting

$$[[A_1 \to A_2]] = m([[A_1]], [[A_2]]).$$

Now if M is an STT-term then there is a context Γ and a type A such that $\Gamma \;\vdash\!\!\blacktriangleright\; _{STT}M : A$ So, by the previous theorem $\Gamma \models M : A$ so that, as every variable is in each X that is in $\mathcal{C}$, we get that $M \in [[A]]$ so that $M \in S$.

■

4.4 Proof of the Normalisation Theorem for STT

Definition 4.9 *For sets Y, S of terms we say that Y* **is S-saturated** *if*

$$M[N/x]N_1 \cdots N_k \in Y \implies (\lambda x.M)NN_1 \cdots N_k \in Y$$

for all $x \in Var$, $k \geq 0$ and all terms $M, N, N_1, \ldots, N_k$ such that $N \in S$. Let

$$S - SAT = \{Y \mid Y \text{ is } S\text{-saturated}\}.$$

Lemma 4.10 *If $Y \in S - SAT$ then for any set X of terms*

1. $(X \to Y) \in S - SAT$,
2. $X \subseteq S \implies (X \to^w Y) \subseteq (X \to Y)$.

For each set X of terms let S^* be the set of terms that have the form

$$xN_1 \cdots N_k$$

where $x \in Var$, $k \geq 0$ and $N_1, \ldots, N_k \in S$.

Definition 4.11 *Call a set S of terms a* **regular set** *if the following conditions hold.*

1. $S^* \subseteq S$,
2. *If $Mx \in S$, with $x \notin \mathsf{var}(M)$ then $M \in S$,*
3. $S \in S - SAT$.

Definition 4.12 *Call (S_0, S)* **pre-adapted** *if the following conditions hold.*

1. $S_0 \subseteq S \subseteq \mathcal{T}$,
2. $S_0 \subseteq (S \to S_0)$,
3. $(S_0 \to S) \subseteq S$.

Lemma 4.13 *If (S_0, S) is pre-adapted then*

$$S_0 \subseteq X, Y \subseteq S \implies S_0 \subseteq (X \to Y) \subseteq S.$$

Proof: An easy consequence of the Lemma 4.4.

■

Lemma 4.14 (Regular Set Lemma) *If S is a regular set of terms then*

1. *(S^*, S) is pre-adapted,*

2. *The collection*

$$\mathcal{C}(S) = \{Y \in S - SAT \mid S^* \subseteq Y \subseteq S\}$$

is an STT-collection,

3. *Every STT-term is in S.*

Lemma 4.15 *The set $\mathcal{N}$ of normalisable terms is a regular set.*

The Normalisation Theorem is an immediate consequence of the previous two lemmas.

Exercise 4.16 *The* **left-most redex** *of a non-normal term is that redex that starts furthest to the left among all redexes. The* **left-most contraction** *is the contraction of the left-most redex. A* **left-most reduction** *is a sequence of left-most contractions. Show that every STT-term has a left-most reduction to its normal form. [Hint: Apply the Regular Set Lemma.*