HANDBOOK OF PHILOSOPHICAL LOGIC
2ND EDITION

VOLUME 15

HANDBOOK OF PHILOSOPHICAL LOGIC

2nd Edition

Volume 15

edited by D.M. Gabbay and F. Guenthner

Volume 1 – ISBN 0-7923-7018-X
Volume 2 – ISBN 0-7923-7126-7
Volume 3 – ISBN 0-7923-7160-7
Volume 4 – ISBN 1-4020-0139-8
Volume 5 – ISBN 1-4020-0235-1
Volume 6 – ISBN 1-4020-0583-0
Volume 7 – ISBN 1-4020-0599-7
Volume 8 – ISBN 1-4020-0665-9
Volume 9 – ISBN 1-4020-0699-3
Volume 10 – ISBN 1-4020-1644-1
Volume 11 – ISBN 1-4020-1966-1
Volume 12 – ISBN 1-4020-3091-6
Volume 13 – ISBN 978-1-4020-3520-3
Volume 14 – ISBN 978-1-4020-6323-7

HANDBOOK OF PHILOSOPHICAL LOGIC

2nd EDITION

VOLUME 15

Edited by

D.M. GABBAY
King's College, London, U.K.

and

F. GUENTHNER
Centrum für Informations- und Sprachverarbeitung,
Ludwig-Maximilians-Universität München, Germany

Springer

Editors
Prof. Dr. Dov M. Gabbay
King's College London
Dept. Computer Science
London
United Kingdom
dov.gabbay@kcl.ac.uk

Franz Guenthner
University of Munich
Centrüm für Informations- und Sprachverarbeitung
Oettingenstr. 67
80538 Munich
Germany
gue@cis.uni-muenchen.de

ISBN 978-94-007-0484-8 e-ISBN 978-94-007-0485-5
DOI 10.1007/978-94-007-0485-5
Springer Dordrecht Heidelberg London New York

CONTENTS

Editorial Preface vii
Dov M. Gabbay

Lambda Calculi: A Guide 1
Chris Hankin

Interpolation and Definability 67
Dov M. Gabbay and Larisa L. Maksimova

Discourse Representation Theory 125
Hans Kamp, Josef van Genabith and Uwe Reyle

Index 395

PREFACE TO THE SECOND EDITION

It is with great pleasure that we are presenting to the community the second edition of this extraordinary handbook. It has been over 15 years since the publication of the first edition and there have been great changes in the landscape of philosophical logic since then.

The first edition has proved invaluable to generations of students and researchers in formal philosophy and language, as well as to consumers of logic in many applied areas. The main logic article in the Encyclopaedia Britannica 1999 has described the first edition as 'the best starting point for exploring any of the topics in logic'. We are confident that the second edition will prove to be just as good!

The first edition was the second handbook published for the logic community. It followed the North Holland one volume *Handbook of Mathematical Logic*, published in 1977, edited by the late Jon Barwise. The four volume *Handbook of Philosophical Logic*, published 1983–1989 came at a fortunate temporal junction at the evolution of logic. This was the time when logic was gaining ground in computer science and artificial intelligence circles.

These areas were under increasing commercial pressure to provide devices which help and/or replace the human in his daily activity. This pressure required the use of logic in the modelling of human activity and organisation on the one hand and to provide the theoretical basis for the computer program constructs on the other. The result was that the *Handbook of Philosophical Logic*, which covered most of the areas needed from logic for these active communities, became their bible.

The increased demand for philosophical logic from computer science and artificial intelligence and computational linguistics accelerated the development of the subject directly and indirectly. It directly pushed research forward, stimulated by the needs of applications. New logic areas became established and old areas were enriched and expanded. At the same time, it socially provided employment for generations of logicians residing in computer science, linguistics and electrical engineering departments which of course helped keep the logic community thriving. In addition to that, it so happens (perhaps not by accident) that many of the Handbook contributors became active in these application areas and took their place as time passed on, among the most famous leading figures of applied philosophical logic of our times. Today we have a handbook with a most extraordinary collection of famous people as authors!

The table below will give our readers an idea of the landscape of logic and its relation to computer science and formal language and artificial intelligence. It shows that the first edition is very close to the mark of what was needed. Two topics were not included in the first edition, even though

D. Gabbay and F. Guenthner (eds.),
Handbook of Philosophical Logic, Volume 15, vii–ix.
DOI 10.1007/978-94-007-0485-5, © Springer Science+Business Media B.V. 2011

they were extensively discussed by all authors in a 3-day Handbook meeting. These are:

- a chapter on non-monotonic logic
- a chapter on combinatory logic and λ-calculus

We felt at the time (1979) that non-monotonic logic was not ready for a chapter yet and that combinatory logic and λ-calculus was too far removed.[1] Non-monotonic logic is now a very major area of philosophical logic, alongside default logics, labelled deductive systems, fibring logics, multi-dimensional, multimodal and substructural logics. Intensive reexaminations of fragments of classical logic have produced fresh insights, including at time decision procedures and equivalence with non-classical systems.

Perhaps the most impressive achievement of philosophical logic as arising in the past decade has been the effective negotiation of research partnerships with fallacy theory, informal logic and argumentation theory, attested to by the Amsterdam Conference in Logic and Argumentation in 1995, and the two Bonn Conferences in Practical Reasoning in 1996 and 1997.

These subjects are becoming more and more useful in agent theory and intelligent and reactive databases.

Finally, fifteen years after the start of the Handbook project, I would like to take this opportunity to put forward my current views about logic in computer science, computational linguistics and artificial intelligence. In the early 1980s the perception of the role of logic in computer science was that of a specification and reasoning tool and that of a basis for possibly neat computer languages. The computer scientist was manipulating data structures and the use of logic was one of his options.

My own view at the time was that there was an opportunity for logic to play a key role in computer science and to exchange benefits with this rich and important application area and thus enhance its own evolution. The relationship between logic and computer science was perceived as very much like the relationship of applied mathematics to physics and engineering. Applied mathematics evolves through its use as an essential tool, and so we hoped for logic. Today my view has changed. As computer science and artificial intelligence deal more and more with distributed and interactive systems, processes, concurrency, agents, causes, transitions, communication and control (to name a few), the researcher in this area is having more and more in common with the traditional philosopher who has been analysing

[1] I am really sorry, in hindsight, about the omission of the non-monotonic logic chapter. I wonder how the subject would have developed, if the AI research community had had a theoretical model, in the form of a chapter, to look at. Perhaps the area would have developed in a more streamlined way!

such questions for centuries (unrestricted by the capabilities of any hardware).

The principles governing the interaction of several processes, for example, are abstract an similar to principles governing the cooperation of two large organisation. A detailed rule based effective but rigid bureaucracy is very much similar to a complex computer program handling and manipulating data. My guess is that the principles underlying one are very much the same as those underlying the other.

I believe the day is not far away in the future when the computer scientist will wake up one morning with the realisation that he is actually a kind of formal philosopher!

The projected number of volumes for this Handbook is about 18. The subject has evolved and its areas have become interrelated to such an extent that it no longer makes sense to dedicate volumes to topics. However, the volumes do follow some natural groupings of chapters.

I would like to thank our authors are readers for their contributions and their commitment in making this Handbook a success. Thanks also to our publication administrator Mrs J. Spurr for her usual dedication and excellence and to Kluwer Academic Publishers for their continuing support for the Handbook.

Dov M. Gabbay
King's College London

Logic	**IT**			
	Natural language processing	**Program control specification, verification, concurrency**	**Artificial intelligence**	**Logic programming**
Temporal logic	Expressive power of tense operators. Temporal indices. Separation of past from future	Expressive power for recurrent events. Specification of temporal control. Decision problems. Model checking.	Planning. Time dependent data. Event calculus. Persistence through time—the Frame Problem. Temporal query language. temporal transactions.	Extension of Horn clause with time capability. Event calculus. Temporal logic programming.
Modal logic. Multi-modal logics	generalised quantifiers	Action logic	Belief revision. Inferential databases	Negation by failure and modality
Algorithmic proof	Discourse representation. Direct computation on linguistic input	New logics. Generic theorem provers	General theory of reasoning. Non-monotonic systems	Procedural approach to logic
Non-monotonic reasoning	Resolving ambiguities. Machine translation. Document classification. Relevance theory	Loop checking. Non-monotonic decisions about loops. Faults in systems.	Intrinsic logical discipline for AI. Evolving and communicating databases	Negation by failure. Deductive databases
Probabilistic and fuzzy logic	logical analysis of language	Real time systems	Expert systems. Machine learning	Semantics for logic programs
Intuitionistic logic	Quantifiers in logic	Constructive reasoning and proof theory about specification design	Intuitionistic logic is a better logical basis than classical logic	Horn clause logic is really intuitionistic. Extension of logic programming languages
Set theory, higher-order logic, λ-calculus, types	Montague semantics. Situation semantics	Non-well-founded sets	Hereditary finite predicates	λ-calculus extension to logic programs

Imperative vs. declarative languages	**Database theory**	**Complexity theory**	**Agent theory**	**Special comments: A look to the future**
Temporal logic as a declarative programming language. The changing past in databases. The imperative future	Temporal databases and temporal transactions	Complexity questions of decision procedures of the logics involved	An essential component	Temporal systems are becoming more and more sophisticated and extensively applied
Dynamic logic	Database updates and action logic	Ditto	Possible actions	Multimodal logics are on the rise. Quantification and context becoming very active
Types. Term rewrite systems. Abstract interpretation	Abduction, relevance	Ditto	Agent's implementation rely on proof theory.	
	Inferential databases. Non-monotonic coding of databases	Ditto	Agent's reasoning is non-monotonic	A major area now. Important for formalising practical reasoning
	Fuzzy and probabilistic data	Ditto	Connection with decision theory	Major area now
Semantics for programming languages. Martin-Löf theories	Database transactions. Inductive learning	Ditto	Agents constructive reasoning	Still a major central alternative to classical logic
Semantics for programming languages. Abstract interpretation. Domain recursion theory.		Ditto		More central than ever!

Classical logic. Classical fragments	Basic background language	Program synthesis	A basic tool	
Labelled deductive systems	Extremely useful in modelling		A unifying framework. Context theory.	Annotated logic programs
Resource and substructural logics	Lambek calculus		Truth maintenance systems	
Fibring and combining logics	Dynamic syntax	Modules. Combining languages	Logics of space and time	Combining features
Fallacy theory				
Logical Dynamics	Widely applied here			
Argumentation theory games		Game semantics gaining ground		
Object level/ metalevel			Extensively used in AI	
Mechanisms: Abduction, default relevance			ditto	
Connection with neural nets				
Time-action-revision models			ditto	

	Relational databases	Logical complexity classes	The workhorse of logic	The study of fragments is very active and promising.
	Labelling allows for context and control.		Essential tool.	The new unifying framework for logics
Linear logic			Agents have limited resources	
	Linked databases. Reactive databases		Agents are built up of various fibred mechanisms	The notion of self-fibring allows for self-reference
				Fallacies are really valid modes of reasoning in the right context.
			Potentially applicable	A dynamic view of logic
				On the rise in all areas of applied logic. Promises a great future
			Important feature of agents	Always central in all areas
			Very important for agents	Becoming part of the notion of a logic
				Of great importance to the future. Just starting
			A new theory of logical agent	A new kind of model

CHRIS HANKIN

LAMBDA CALCULI: A GUIDE

1 INTRODUCTION

One of the universal notions of programming languages is functional abstraction. The methods of Java and the functions defined and used in functional programming languages, such as Haskell, are instances of this general notion. The inspiration for this form of abstraction mechanism comes from Mathematical Logic; notably Church's λ(lambda)-calculi and Schönfinkel's and Curry's Combinatory Logic. A proper study of these foundations leads to a better understanding of some of the fundamental issues in Computer Science. Areas in which they have had a major influence include:

Programming Language Design: We have already suggested the link with the notion of functional abstraction in programming languages. In addition, many of the typing notions found in modern programming languages have been inspired by the typing mechanisms found in these formal calculi. A notable example of this, to which we shall return, is the style of *polymorphism* which is employed in modern functional programming languages.

Semantics: One of the predominant schools of thought on this topic is *denotational semantics*. In this approach a typed λ-calculus is used as the meta-language; the meaning of a program is expressed by mapping it into a corresponding λ-calculus object. Understanding what such objects are requires that we should have a *model* of the calculus; the construction of such models has been the motivation for the subject of *domain theory*.

Computability: A classical use of the λ-calculus was in the study of computability; the study of the theoretical limitations of formal systems for describing computations. Indeed the first result in computability was a result concerning the relationship between the λ-calculus and Kleene's Recursive Functions. The (un-)decidability results familiar from automata theory have their analogues in the theory of the λ-calculus.

Natural Language Understanding: Since Richard Montague's pioneering use of λ-abstractions in natural language semantics in the 1970s there has been extensive use of extended calculi within the linguistics community.

I hope that this chapter will serve as an introduction to the λ-calculus for students of these areas.

D. Gabbay and F. Guenthner (eds.),
Handbook of Philosophical Logic, Volume 15, 1–66.
DOI 10.1007/978-94-007-0485-5_1, © Springer Science+Business Media B.V. 2011

2 SYNTAX

Classically, in set theory, a function is represented by its *graph*. The graph of a function defines a function by its input/output behaviour; for example, a unary function is represented by a set of pairs where the first component of each pair specifies the argument and the second component specifies the corresponding result. From this perspective, the function on pairs of natural numbers which adds its two arguments is represented as:

$$\{((0,0),0),((0,1),1),\ldots,((1,0),1),((1,1),2),\ldots\}$$

or:

$$\{((m,n),p) \mid m,n \in Num, p = n+m\}$$

Two functions are equal if they have the same graph; this notion of equality is referred to as *extensional* equality and we will return to it later.

From the point of view of Computer Science, this representation is not very useful. We are usually as interested in *how* a function computes its answer as in *what* it computes. For example, all sorting functions have the same graph and are thus (extensionally) equal but a significant part of the Computer Science literature has been devoted to the definition and analysis of different sorting algorithms, so we are clearly missing something. The casual use of the word "algorithm" in the last sentence is the key; we should represent a function by a rule, which describes how the result is calculated, rather than its graph. In this scheme, two functions are equal if they are both defined by the same (or equivalent) rules; this form of equality is called *intensional* equality. The λ-calculus[1] provides a formalism for expressing functions as *rules of correspondence* between arguments and results.

The λ-calculus consists of a notation for expressing rules, λ-notation, and a set of axioms and rules which tell us how to compute with terms expressed in the notation. A BNF specification of the λ-notation is:

$<\lambda$-term>	::=	<variable> |	
		$(\lambda$ <variable>$<\lambda$-term>) |	**(abs)**
		$(<\lambda$-term>$<\lambda$-term>)	**(app)**
<variable>	::=	$x \mid y \mid z \ldots$	

It is more usual to present the syntax of a formal calculus using an inductive definition. The λ-calculus may defined in this style in the following

[1] There is a wide variety of different λ-calculi. The calculi differ along many axes: syntax, typing, rules of inference,.... When we talk of the λ-calculus we generally mean the pure, type-free $\lambda K\beta$-calculus which is the primary object of study in Barendregt's encyclopaedic book.

way. The class of λ-terms consists of words constructed from the following alphabet:

$x, y, z, \ldots$	variables
λ	
$(\,,\,)$	parentheses

We define terms formally as follows:

DEFINITION 1 (λ-terms).
The class Λ of λ-terms is the least class satisfying the following:

1. $x \in \Lambda$, x a variable
2. if $M \in \Lambda$ then $(\lambda x M) \in \Lambda$
3. if $M, N \in \Lambda$ then $(MN) \in \Lambda$

We specify the least class to avoid including "junk" terms; the clauses of the definition say what has to be in the class, not what shouldn't be. Some λ-terms are:

$$x \qquad (xz) \qquad ((xz)(yz)) \qquad (\lambda x(\lambda y(\lambda z((xz)(yz)))))$$

The intuition is that terms matching clause 2 correspond to function definitions, where the variable after the λ specifies the name of the formal parameter, and terms matching clause 3 correspond to function applications. Thus:

$$(\lambda x x)$$

should be compared to:

```
(\ x -> x)
```

in Haskell, or to:

```
public int id(int x) { return x;}
```

in Java[2]. To avoid the proliferation of parentheses, we will generally use an alternative notation for terms constructed according to clause 2 of the definition:

$$\lambda x.M$$

[2]The λ-term is type-free. This is in contrast to the Haskell function which is polymorphic and the Java method which is strongly (monomorphically) typed. Both the λ-term and the Haskell program are actually equivalent to a whole set of Java methods with an element for every type.

and moreover, we will elide internal λs and "."s and assume that abstraction associates to the right so that the following terms are equivalent:

$$\lambda x_1 \ldots x_n.M \equiv \lambda \vec{x}.M \equiv (\lambda x_1(\ldots(\lambda x_n M)\ldots))$$

where $\vec{x}$ is our notation for the sequence $x_1, \ldots, x_n$. We will generally use the symbol $\equiv$ to denote syntactic equality between terms.

The symbol λ acts as a variable binder in a similar fashion to $\int \ldots dx$ in integral calculus and the quantifiers $\exists$ and $\forall$ in predicate calculus. The set of bound variables is defined inductively by the following function, $BV : \Lambda \to \mathcal{P}(Var)$[3]:

$$\begin{array}{lcl} BV\, x & = & \emptyset \\ BV\,(\lambda x M) & = & (BV\, M) \cup \{x\} \\ BV\,(MN) & = & (BV\, M) \cup (BV\, N) \end{array}$$

We will also need the set of free variables in a term; these are defined inductively by the following function, $FV : \Lambda \to \mathcal{P}(Var)$:

$$\begin{array}{lcl} FV\, x & = & \{x\} \\ FV\,(\lambda x M) & = & (FV\, M) - \{x\} \\ FV\,(MN) & = & (FV\, M) \cup (FV\, N) \end{array}$$

When $(FV\, M)$ is the empty set, $\emptyset$, M is said to be *closed*; closed terms are sometimes called *combinators* and the class of all such terms is Λ^0. Notice that the sets of bound and free variables are not necessarily disjoint; x occurs both bound and free in:

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

Terms which are defined by clause 3 of the definition correspond to applications. We adopt the convention that application is left associative. Consequently:

$$MN_1 \ldots N_n \equiv M\vec{N} \equiv (\ldots(MN_1)\ldots N_n)$$

In the following, we also make use of the notion of subterm. A *subterm* of a λ-term is some part of the term which is itself a well-formed λ-term; we can generate the set of subterms using the function, $Sub : \Lambda \to \mathcal{P}(\Lambda)$, defined as follows:

$$\begin{array}{lcl} Sub\, x & = & \{x\} \\ Sub\,(\lambda x M) & = & (Sub\, M) \cup \{(\lambda x M)\} \\ Sub\,(MN) & = & (Sub\, M) \cup (Sub\, N) \cup \{(MN)\} \end{array}$$

[3] $f : A \to B$ means that f is a function which takes arguments from the "set" A to results in B. The notation $\mathcal{P}(A)$ constructs the powerset of A.

Although we have presented an recursive definition of subterms, we could think of "subterm" as being a reflexive, transitive, binary relation on terms. Our definition does not distinguish between different occurrences of the same subterm; to do so we would need to construct a multi-set of subterms but we will not require this refinement in the following.

Often, we will need the notion of a partially specified term, that is a term with "holes" in it. Such a term gives a context into which we can put other terms (to fill the holes!). The ability to construct contexts will clarify some definitions and generalise some results (for example see the generalisation of the Substitution Lemma later). We give an inductive definition of contexts for λ-terms:

DEFINITION 2 (Contexts).
The class $\mathcal{C}[]$ of λ-contexts is the least class satisfying:

1. $x \in \mathcal{C}[]$
2. $[] \in \mathcal{C}[]$
3. if $C_1[], C_2[] \in \mathcal{C}[]$ then $(C_1[]C_2[]), (\lambda x C_1[]) \in \mathcal{C}[]$

Notice that a hole is represented by $[]$. An example of a context is:

$$((\lambda x.[]x)M)$$

We will often give a name to a context, say $C[]$ for the one above, such names will always terminate with "$[]$". To represent the term generated by filling the holes in a context with some term, we write the name of the context with the term that is to fill the hole appearing between the square brackets. Thus:

$$C[\lambda y.y]$$

is the term:

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

Of course, a context may have many holes but they will all be filled with the same term; we could generalise this by labelling holes, in which case different holes could be filled by different terms, but we will not need such generality in here. Notice that variables in $FV(M)$ might become bound in $C[M]$ if the context has holes inside λ-abstractions.

A major limitation of the notation seems to be that we can only define unary functions; we introduce one formal argument at a time. The fact that this is not a real restriction was first observed by Schönfinkel. Given some binary function denoted by an expression in formal arguments x and y, say $f(x, y)$, then we define:

$$a \equiv (\lambda y(\lambda x(f(x, y))))$$

then a is equivalent to the original function but takes its arguments one at a time[4].

We next introduce the basic theory of equality between λ-terms. We will identify a set of "canonical" terms – the *normal* forms. We present material on reduction; this concerns how we compute with terms. Next we turn to semantics and abstractly characterise models of the λ-calculus. We then consider the relationship between the λ-calculus and other notions of computable functions. Finally we present a number of typed calculi.

3 THE BASIC THEORY

3.1 The theory λ

We can construct formulae from the terms; a theory then establishes certain formulae as axioms and provides rules of inference which enable us to derive new formulae. The true formulae (either axioms or formulae that can be derived from the rules) are called *theorems*.

We now present a theory of equality (or *convertibility*) between λ-terms. There are a number of reasonable requirements for such a theory:

1. An application term should be equal to the result obtained by applying the function part of the term to the argument. For example, suppose that Java methods can be higher-order (take methods as arguments and produce them as results) and that we have defined a higher-order variant of `id`. Then:

   ```
   id(fun)
   ```

   should surely be the same method as `fun` (for any appropriate method parameter `fun`).

2. Equality should be an equivalence relation.

3. Equal terms should be equal in any context.

These requirements go some way to motivating the theory λ which is shown in Figure 1.

The rule (ξ) is sometimes called the rule of weak extensionality. The rule (β) is the rule which corresponds to function application. The notation $M[x := N]$ should be read "replace free occurrences of x in M by N" (some care must be taken — we return to this later). The classical presentation of the theory also includes an α-rule which allows a change of bound variable names. Computer Science readers should compare the rule (β) to

[4] A function such as a, which takes its arguments one at a time, is often called a *curried* function (in honour of the logician Haskell B. Curry).

$$(\lambda x.M)N = M[x := N] \qquad (\beta)$$

$$M = M$$

$$\frac{M = N}{N = M}$$

$$\frac{M = N \quad N = L}{M = L}$$

$$\frac{M = N}{MZ = NZ}$$

$$\frac{M = N}{ZM = ZN}$$

$$\frac{M = N}{\lambda x.M = \lambda x.N} \qquad (\xi)$$

Figure 1. The theory λ

their intuitive understanding of the meaning of procedure calls in a familiar programming language.

We write:

$$\lambda \vdash M = N$$

to mean that $M = N$ is a theorem of λ and read the theorem as "M and N are convertible". The notation of λ-terms and this theory are variously called the λ-calculus (the name that we will use in the following), the $\lambda\beta$-calculus, the λK-calculus or the $\lambda K\beta$-calculus.

Note that:

$$M \equiv N \Rightarrow M = N$$

but:

$$\neg(M = N \Rightarrow M \equiv N)$$

For example:

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

but the two terms are not identical.

Finally, we illustrate the use of the theory to prove a fundamental theorem, the Fixed Point Theorem. This will play an important role when we consider computability. Fixed points give meaning to self-referential constructs such as recursive method definitions or functions. The theorem fixes a term F and then states that there is another term which is a fixed point for F. The proof of the theorem is constructive in that it shows precisely how to construct the required term.

THEOREM 3 (The Fixed Point Theorem).

$$\forall F \in \Lambda, \exists X \in \Lambda . FX = X$$

Proof.
Let $W \equiv \lambda x.F(xx)$ and $X \equiv WW$. Then

$$X \equiv WW \equiv (\lambda x.F(xx))W = F(WW) \equiv FX$$

■

In a more familiar context, for example, 1 is a fixed point of the squaring function. The Fixed Point Theorem may seem quite surprising at first sight; it says that all terms have fixed points. For some terms, such as:

$$\lambda x.x$$

which is the identity function, this is obvious (all terms are fixed points of the identity!) but for others, such as:

$$\lambda xy.xy$$

it is not so clear. However, since the proof of the Fixed Point Theorem is constructive; it gives a recipe for constructing a fixed point of any term. In the second case above this leads to the following construction:

$$W \equiv \lambda x.(\lambda xy.xy)(xx) = \lambda x.\lambda y.(xx)y \equiv \lambda xy.(xx)y$$

The required fixed point is thus

$$(\lambda xy.(xx)y)(\lambda xy.(xx)y)$$

we can check that this is indeed a fixed point of the original term:

$$\begin{array}{l}(\lambda xy.(xx)y)(\lambda xy.(xx)y)\\ = \lambda y.((\lambda xy.(xx)y)(\lambda xy.(xx)y))y\\ = (\lambda xy.xy)((\lambda xy.(xx)y)(\lambda xy.(xx)y))\end{array}$$

The fixed point constructed for the identity function is:

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

This term plays a special role in the theory which we shall return to later[5].

[5]For those readers familiar with domain theory, this term plays the same role as $\perp$. It is the least fixed point of the identity function (and many others!).

Fixed points are important in Computer Science. They play a fundamental role in the semantics of recursive definitions. For example, the factorial function:

$$\begin{array}{lcl} fac\ 0 & = & 1 \\ fac\ (succ\ n) & = & (succ\ n) \times (fac\ n) \end{array}$$

is a fixed point of the term:

$$\lambda fn.if(=\ n\ 0)\ 1\ (\times\ n\ (f(pred\ n)))$$

(of course we must be careful about reading too much into this term – $0, 1, \times$ are just formal symbols, variables, they have no deeper significance in the λ-calculus which we have defined so far). We shall return to this point later.

3.2 Substitution

We now return to the substitution operation used in the rule (β). A naive approach to defining this operation leads to the problem of "variable capture". This problem occurs when we naively substitute a term containing a free variable into a scope where the variable becomes bound. For example:

$$(\lambda xy.yx)y \neq \lambda y.yy$$

The free occurrence of y in the left hand term is analogous to a global variable in programming, in the right hand side the global variable has become confused with the bound variable (formal parameter). We will consider three different approaches to this problem before selecting one for use in the rest of this article.

Three Approaches

The Classical Approach: The first approach is based on Church's original treatment of substitution. We use the following definition:

1. $x[x := N] \equiv N$
2. $y[x := N] \equiv y$, if x is not the same as y
3. $(\lambda x.M)[x := N] \equiv \lambda x.M$
4. $(\lambda y.M)[x := N] \equiv \lambda y.M[x := N]$, if $x \notin FV\ M$ or $y \notin FV\ N$
5. $(\lambda y.M)[x := N] \equiv \lambda z.(M[y := z])[x := N]$, if $x \in FV\ M$ and $y \in FV\ N$, z a new variable
6. $(M_1 M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])$

We consider the three rules 3 to 5 in a bit more detail. Rule 3 applies when the variable being substituted for is bound at the outermost level; in this case there will be no free occurrences of x in the remainder of the expression and thus the substitution has no effect. Rule 4 is applicable when variable capture cannot occur, either x does not occur free in the body (in which case the substitution is a no-operation again) or the variable that is bound in the outermost level does not occur free in the term being substituted (no capture); in either case the substitution can be pushed through the λ to apply to the body. Rule 5 applies when variable capture could occur, that is when some substitution does take place and the variable bound at the outermost level does occur free in the term being substituted; in this case, we first rename the bound variable to a completely new variable.

Rule 5 is only valid under the assumption that terms which are similar, having the same free variables and differing only in their bound variables, are essentially the same. This is reasonable if we think about programming languages:

```
public int id(int y) { return y;}
```

the above method is clearly the same as the earlier one with the same name; we have only changed the formal parameters. In Church's original presentation of the λ-calculus there were two additional axioms; (α) formalises the above discussion and (η) introduces extensional equality (see below). The alpha rule is:

$$\lambda x.M = \lambda y.M[x := y], y \notin FV\ M \qquad (\alpha)$$

The Variable Convention: For our second definition of the substitution operation, which is introduced in Barendregt's book, we start with two definitions:

DEFINITION 4 (Change of Bound Variables).
M' is produced from M by a *change of bound variables* if $M \equiv C[\lambda x.N]$ and $M' \equiv C[\lambda y.(N[x := y])]$ where y does not occur at all in N and $C[]$ is a context with one hole.

DEFINITION 5 (α-congruence).
M is α-congruent to N, written $M \equiv_\alpha N$, if N results from M by a series of changes of bound variable.

According to the second definition, we have:

$$\lambda x.xy \equiv_\alpha \lambda z.zy$$

but not:

$$\lambda x.xy \equiv_\alpha \lambda y.yy$$

Notice that the first two terms are also equal by the rule (α) but the second two are not. Our strategy for defining substitution is as follows:

1. Identify α-congruent terms
2. Consider a λ-term as a representative of its equivalence class
3. Interpret $M[x := N]$ as an operation on the equivalence classes, using representatives according to the following *variable convention*:

DEFINITION 6 (Variable Convention).
If $M_1, \ldots, M_n$ occur in a certain context then in these terms all bound variables are chosen to be different from free variables[6].

With this strategy, we can define substitution as follows:

1. $x[x := N] \equiv N$
2. $y[x := N] \equiv y$, if $x \not\equiv y$
3. $(\lambda y.M)[x := N] \equiv \lambda y.(M[x := N])$
4. $(M_1 M_2)[x := N] \equiv (M_1[x := N])(M_2[x := N])$

The variable capture problem has disappeared! — the reason for this is that for y to appear free in N in the context:

$$(\lambda y.M)[x := N]$$

would breach the variable convention so we would have to use a different representative of the α-equivalence class of $\lambda y.M$ (this is precisely what rule 5 in the classical approach makes explicit). In the following, we will adopt this convention and definition of substitution because it is easier to work with (there are less cases to consider in proofs). An example of its use is illustrated below:

$$\begin{array}{rcll} (\lambda xyz.xzy)(\lambda xz.x) & = & \lambda yz.(\lambda xw.x)zy & \text{by the variable convention} \\ & = & \lambda yz.(\lambda w.z)y & \\ & = & \lambda yz.z & \end{array}$$

However, before continuing we consider a third alternative.

[6] We have already implicitly employed this convention in the proof of the Fixed Point Theorem — consider what happens if x occurs free in the term F.

The de Bruijn Notation: The third approach to defining substitution avoids the problem of variable capture by banishing free variables. We revise the definition of λ-terms so that parameters occurring in the body of a term are referred to by natural numbers which uniquely identify the binding λ. For example:

$$\lambda.\lambda.2 \text{ is equivalent to } \lambda xy.x$$

This is the notation invented by de Bruijn and used in the Automath project, an automated theorem proving system. More formally the terms in de Bruijn's notation are defined inductively as the least set such that:

1. any natural number (greater than zero) is a term
2. If M and N are terms, then (MN) is a term
3. If M is a term, (λM) is a term

and (β) is replaced by:

$$(\lambda P)Q = P[1 := Q]$$

where:

$$\begin{array}{lcl} n[m := N] & \equiv & \begin{cases} n & \text{if } n < m \\ n-1 & \text{if } n > m \\ rename_{n,1}(N) & \text{if } n = m \end{cases} \\ (M_1 M_2)[m := N] & \equiv & (M_1[m := N])(M_2[m := N]) \\ (\lambda M)[m := N] & \equiv & \lambda(M[m+1 := N]) \end{array}$$

and

$$\begin{array}{lcl} rename_{m,i}(j) & \equiv & \begin{cases} j & \text{if } j < i \\ j+m-1 & \text{if } j \geq i \end{cases} \\ rename_{m,i}(N_1 N_2) & \equiv & rename_{m,i}(N_1) rename_{m,i}(N_2) \\ rename_{m,i}(\lambda N) & \equiv & \lambda(rename_{m,i+1}(N)) \end{array}$$

The following example illustrates the effect of this new β rule:

EXAMPLE 7.

$$\begin{array}{rcl} \lambda.(\lambda.\lambda.2)1 & = & \lambda.(\lambda.2)[1 := 1] \\ & \equiv & \lambda.\lambda.2[2 := 1] \\ & \equiv & \lambda.\lambda.rename_{2,1}(1) \\ & \equiv & \lambda.\lambda.2 \end{array}$$

(c.f. $(\lambda x.(\lambda yz.y)x)$).

Notice the rôle that *rename* takes in relabelling variable indices. There is a simple translation between standard λ-terms and de Bruijn terms (notice that α-congruent terms are equal in the de Bruijn notation):

$$\begin{array}{lcl} DB\, x\ (x_1, \ldots, x_n) & = & i, \text{ if } i \text{ is the minimum such that } x \equiv x_i \\ DB\, (\lambda x M)\ (x_1, \ldots, x_n) & = & \lambda(DB\, M\ (x, x_1, \ldots, x_n)) \\ DB\, (MN)\ (x_1, \ldots, x_n) & = & (DB\, M\ (x_1, \ldots, x_n))(DB\, N\ (x_1, \ldots, x_n)) \end{array}$$

The de Bruijn notation is not very readable but the beta rule is easy to implement; indeed it inspired the Categorical Abstract Machine — an efficient mechanism for the implementation of functional languages.

The Substitution Lemma

From now on, we will assume the variable convention unless otherwise stated.
We now present a result which allows us to reorder substitutions, The Substitution Lemma.

LEMMA 8 (The Substitution Lemma).
If x and y are distinct variables and $x \notin FV\ L$ then

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]]$$

The proof is a straightforward induction on the structure of M.

Substitution has a number of other useful properties with respect to convertibility:

1. $M = M' \Rightarrow M[x := N] = M'[x := N]$
2. $N = N' \Rightarrow M[x := N] = M[x := N']$
3. $M = M', N = N' \Rightarrow M[x := N] = M'[x := N']$

These properties are useful but care should be taken when applying them. A major property of functional languages is *referential transparency*; the property which allows equals to be substituted by equals. The properties of substitution appear to be related to this concept but referential transparency is more. For example, the following inference does not follow from (1) to (3):

$$N = N' \Rightarrow \lambda x.x(\lambda y.N) = \lambda x.x(\lambda y.N')$$

This is because we cannot express the two sides of the second equality in the correct form:

$$\lambda x.x(\lambda y.N) \text{ is not the same as } (\lambda x.x(\lambda y.z))[z := N]$$

since N may contain free occurrences of y. The correct formulation of the property of referential transparency, also referred to as Leibniz Law, is:

LEMMA 9 (Referential Transparency).
Let $C[]$ be a context, then

$$N = N' \Rightarrow C[N] = C[N']$$

The proof is by induction on the structure of contexts. The variable convention takes care that we do not inadvertently capture any variables in this substitution.

3.3 Extensionality

The convertibility relationship, $=$, is intensional equality; two terms are equal if they encode the same algorithm in some sense. This does not equate some terms which we might naturally think of as equal. For example, consider a term which has one bound variable and applies some constant term (i.e. a term that does not contain free occurrences of the bound variable) to any term bound to the variable:

$$\lambda x.Mx$$

this term should surely be equal to M since if we apply either $\lambda x.Mx$ or M to some term N, we end up with MN. The formula:

$$\lambda x.Mx = M$$

is not a theorem of λ; there are two ways we can extend λ to make the above formula a theorem. Firstly, we could add a new rule to the theory, giving the new theory $\lambda + ext$:

$$\frac{Mx = Nx}{M = N} \qquad x \notin (FV\ MN) \qquad (\mathbf{ext})$$

Alternatively, we can add a new axiom, giving the new theory $\lambda\eta$ (as proposed by Church):

$$\lambda x.Mx = M, x \notin FV\ M \qquad (\eta)$$

In fact, the following result can be shown:

LEMMA 10. $\lambda + \mathbf{ext}$ *and* $\lambda\eta$ *are equivalent*

The calculus based on $\lambda\eta$ or $\lambda + \mathbf{ext}$ is alternatively called the $\lambda\eta$-calculus, the $\lambda\beta\eta$-calculus, the $\lambda K\eta$-calculus or the $\lambda K\beta\eta$-calculus. Practically, from the point of view of functional programming, the $\lambda\eta$-calculus is not as important as the $\lambda\beta$-calculus since the rule (η) is not normally implemented. The term $\lambda x.Mx$ is a weak head normal form and is thus distinguishable from M; the former is a "value" whilst the latter may lead to a non-terminating computation. Even in an eager language, such as Standard ML, the two terms are distinguished. However, the $\lambda\eta$-calculus does have some theoretical significance which we shall return to later.

3.4 Consistency and Completeness

For a theory to be useful, there must be some theorems and not all closed formulae should be theorems. The former is satisfied provided that the theory has at least one axiom. The latter is slightly trickier and is quite a

fragile property; a theory which satisfies this constraint is called *consistent*. Both of the theories presented here are consistent but it is very easy to lose consistency as we shall see.

We start by formalising the concept. First, some definitions:

DEFINITION 11. An *equation* is a formula of the form:

$$M = N$$

where $M, N \in \Lambda$.

DEFINITION 12. An equation is *closed* if $M, N \in \Lambda^0$.

DEFINITION 13 (Consistency).
If $\mathcal{T}$ is a theory with equations as formulae then $\mathcal{T}$ is *consistent*, written $\mathrm{Con}(\mathcal{T})$, if it does not prove every closed equation.
If $\mathcal{T}$ is a set of equations then $\lambda + \mathcal{T}$ is formed by adding the equations of $\mathcal{T}$ as axioms to λ. $\mathcal{T}$ is consistent, also written $\mathrm{Con}(\mathcal{T})$, if $\mathrm{Con}(\lambda + \mathcal{T})$.

Both of the theories that we have dealt with in this Section, λ and $\lambda\eta$ are consistent (see Barendregt's book). The property of consistency can be lost by adding a single equation. We define the following three terms:

$$\begin{array}{lcl} \mathbf{S} & \equiv & \lambda xyz.xz(yz) \\ \mathbf{K} & \equiv & \lambda xy.x \\ \mathbf{I} & \equiv & \lambda x.x \end{array}$$

Notice that:

$$\begin{array}{lcl} \mathbf{S}MNO & = & MO(NO) \text{ by three applications of } (\beta) \\ \mathbf{K}MN & = & M \text{ by two applications of } (\beta) \\ \mathbf{I}M & = & M \end{array}$$

Now, if we add the equation:

$$\mathbf{S} = \mathbf{K}$$

to either λ or $\lambda\eta$ we get an inconsistent theory. This can be proved as follows (we elide some of the steps):

EXAMPLE 14.

$$\begin{array}{rcl} \mathbf{S} = \mathbf{K} & \Rightarrow & \mathbf{S}ABC = \mathbf{K}ABC \text{ for all } A, B, C \\ & \Rightarrow & AC(BC) = AC \end{array}$$

Now consider the case when $A = C = \mathbf{I}$, then since $\mathbf{I}A = A$ for all A:

$$AC(BC) = AC \Rightarrow B(\mathbf{I}) = \mathbf{I}$$

Now consider the case when $B = \mathbf{K}D$ for some arbitrary D, then:

$$B(\mathbf{I}) = \mathbf{I} \Rightarrow D = \mathbf{I}$$

and thus, since D was arbitrary, all terms are equal to the constant term $\mathbf{I}$.

Consideration of the foregoing motivates the following definition:

DEFINITION 15 (Incompatibility).
Let $M, N \in \Lambda$, then M and N are *incompatible*, written $M\#N$, if $\neg Con(M = N)$.

We now turn to the notion of completeness. Yet again, we start by making some definitions:

DEFINITION 16 (Normal Forms).
If $M \in \Lambda$, then M is a *β-normal form*, written β-nf or nf, if M has no subterms of the form $(\lambda x.R)S$
If $M \in \Lambda$, then M *has a* β-nf if there exists an N such that $N = M$ and N is a β-nf.

Some (non-)examples of normal forms:

$\lambda x.x$ is a nf
$(\lambda xy.x)(\lambda x.x)$ has $\lambda yx.x$ as a nf
$(\lambda x.xx)(\lambda x.xx)$ does not have a nf

By analogy, a $\beta\eta$-nf is a β-nf which also does not contain any subterms of the form:

$$(\lambda x.Rx) \text{ with } x \notin FV\ R$$

We now state the following facts about normal forms:

PROPOSITION 17.

1. *M has a $\beta\eta$-nf $\Leftrightarrow$ M has a β-nf*

2. *If M and N are distinct β-nfs then $M = N$ is not a theorem of λ (and similarly for $\lambda\eta$).*

3. *If M and N are distinct $\beta\eta$-nfs then $M\#N$.*

The use of $\beta\eta$-nfs in the last point is essential; y and $\lambda x.yx$ are distinct β-nfs but not incompatible – they are η-equivalent.

The completeness of $\lambda\eta$ is expressed by the following:

PROPOSITION 18 (Completeness).
Suppose M and N have nfs; then either:
$\lambda\eta \vdash M = N$
or
$\lambda\eta + (M = N)$ *is inconsistent*

4 REDUCTION

Normal forms can be used as canonical representatives for the convertibility equivalence classes. A more computational view results from treating normal forms as the "answers" produced from λ-term "programs". This view is justified by observing that the evaluation of the β-normal form of a term involves removing application subterms by applying the (β) rule; we have already identified this process with function application in programming languages. We will pursue this view further[7].

We illustrate the earlier discussion and motivate the following material by considering an example in a λ-calculus extended with constants. We consider the following program:

```
let
  fac 0  = 1
  fac n = n * fac(n-1)
in fac 0
```

We briefly discussed a variant of this function earlier, where we saw that it was the fixed point of a certain functional. We consider that the calculus which we are using is extended with a constant, $\mathbf{Y}$, which computes the fixed point of a given term; following the construction used in the proof of the Fixed Point Theorem, it is clear that such a constant could be defined by the following term:

$$\mathbf{Y} \equiv \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))$$

The program may be translated in the following way:

$$(\lambda f.f0)(\mathbf{Y}(\lambda fn.if(= n\ 0)1(*\ n\ (f(-\ n\ 1)))))$$

notice that the let-construct has been translated as an application term.

Consider now the normal form of the program. We can produce the normal form by repeatedly applying rule (β); in outline, we perform the

[7]Notice that in lazy functional languages such as Haskell, rather than normal forms, (weak) head normal forms are considered to be answers — we shall return to this point later.

following steps[8]:

$$\begin{array}{rcll}(\lambda f.f0)(\mathbf{Y}\ldots) & = & (\mathbf{Y}\ldots)0 & \\ & = & (\lambda fn.if\ldots)(\mathbf{Y}\ldots)0 & (A)\\ & = & (\lambda n.if(=\ n\ 0)1(*n((\mathbf{Y}\ldots)(-\ n\ 1))))0 & \\ & = & if(=\ 0\ 0)1(*0((\mathbf{Y}\ldots)(-\ 0\ 1))) & \\ & = & if\ \mathbf{true}\ 1\ldots & \\ & = & 1 & \end{array}$$

Throughout this derivation we have used the convertibility relation. Convertibility is symmetrical, indeed it is an equivalence relation, but we have used it in a non-symmetrical way. We are happy to consider 1 as the answer of the above computation, the factorial of 0, but it is a little harder to see the original program as the value of the term "1". The latter view would associate an infinite set of "values" with terms such as "1". We will study some new relations between λ-terms, notably $\rightarrow_\beta$ (one step β-reduction) and $\twoheadrightarrow_\beta$ (β-reduction), the reflexive, transitive closure of $\rightarrow_\beta$. We will see that $\twoheadrightarrow_\beta$ is closely related to $=$ but is not symmetric; each $=$ in the above derivation, other than the one in step (A), could be replaced by $\rightarrow_\beta$.

In performing reduction, we are faced with a problem of strategy. For example, at line (A) there are two subterms of the form $(\lambda x.R)S$ — henceforth called β-redexes (**red**ucible **ex**pression) — as follows:

$$(\lambda fn.if\ldots)(\mathbf{Y}\ldots)0$$

and

$$(\mathbf{Y}\ldots)$$

that is, the whole term and the subterm involving the fixed point combinator. We chose to reduce the first term but consider what would happen if we consistently chose to reduce the fixed point subterm: we would never get to the answer, we would merely construct a larger and larger term! Making the "wrong" choice is not always so catastrophic, for example:

$$\begin{array}{rcl}(\lambda xy./(+\ x\ y)2)((\lambda z.+\ z\ 1)4)6 & \rightarrow_\beta & (\lambda y./(+((\lambda z.+\ z\ 1)4)y)2)6\\ & \rightarrow_\beta & /(+((\lambda z.+\ z\ 1)4)6)2\\ & \rightarrow_\beta & /(+(+\ 4\ 1)6)2\end{array}$$

but also:

[8]We have elided two steps here and used a defining property of fixed point combinators such as $\mathbf{Y}$:

$$\mathbf{Y}F = F(\mathbf{Y}F)$$