Higher Order Logic

Roger Bishop Jones

Abstract

At present this document is a (small) mess pot of explorations of how one might go about presentation of Church's Simple Theory of Types and ultimately ProofPower HOL using Standard ML and/or ProofPower HOL. I am interested both in exposing exactly what Church said, comparing the details of his system with those of ProofPower HOL and discussing the reasons for the differences, but also I am looking for an interesting and digestible way of presenting ProofPower HOL to philosophers or other groups without much IT background. Its doubtful that all these objectives are compatible, and so far my attempts have not been in the least impressive, but I probably will keep picking at it and may eventually come up with something useful.

Created 2010/07/16

Last Change Date: 2011/02/12 09:14:19

http://www.rbjones.com/rbjpub/pp/doc/t038.pdf

Id: t038a.tex,v 1.3 2011/02/12 09:14:19 rbj Exp

Contents

1 Prelude

This document is intended possibly to form a chapter of *Analyses of Analysis* [3]. My initial purpose in preparing the document is simply to present in detail the logical system which is used throughout that work.

Further discussion of what might become of this document in the future may be found in my postscript (Section 9).

In this document, phrases in coloured text are hyperlinks, like on a web page, which will usually get you to another part of this document (the blue parts, the contents list, page numbers in the Index)

but sometimes take you (the red bits) somewhere altogether different (if you happen to be online) like the hist-analytic archives.

2 Introduction

The following presentation of Higher Order Logic is undertaken using the interactive proof assistant ProofPower. The first part of the presentation (Sections 3-5) closely follows Church's paper [2], and the remainder accounts for the subsequent developments which are relevant to an understanding of documents produced using ProofPower.

Church's formulation of the simple theory of types is distinguished by its simplicity, which is made possible by basing the theory upon the typed lambda calculus. This makes it possible to define as higher order constants all but a very few of thee logical connectives.

Because of its great simplicity there is no better place to start than with Church's system. Once this is described the modest adjustments and extensions which help to make STT into a practical language for developing mathematics with a proof tool are readily understood.

It is not always easy for those who have spent a lifetime working with these ideas to understand which parts will prove hard to grasp for people coming to this with a different background. I have assumed some familiarity with modern symbolic logic, and an acquaintance with the idea of programming languages. Beyond that I suspect that the idea of higher order functions, which is pervasive, will be the largest stumbling block for any readers whc are not previously acquainted with it. and so I have made an attempt to make that clear.

I hesitate to say that I anticipate a wide audience for this material, but whatever my expectations on that score, it is written to accompany my work on the application of formal methods using ProofPower to problems in philosophy and elsewhere. It is therefore written with the aim of making it possible for some philosophers to read and understand that kind of formal material, without daring to hope that any other philosopher might adopt the methods.

2.1 Abstract and Concrete Syntax

When Church wrote his paper *Abstract* syntax had not been invented. I first came across this idea when I came to 'denotation semantics', an approach to the semantics of programming languages concocted by a computer scientics (Strachey) and a set theorist (Scott) collaborating at Oxford. It is beneficial when working in the design of complex programming languages, simplifying the task by separating out the details and difficulties of the concrete presentation of the language from the underlying abstract structure both of syntax and of semantics.

In presenting Higher Order Logic here, the emphasis is on the underlying abstract structures. Where examples are given, of necessity they must be presented using some concrete syntax, and this will be the concrete syntax adopted in ProofPower-HOL.

3 Simple Types

Simple types, as presented by Church in his paper presenting *A Formulation of The Simple Theory of Types*[2], consist of the set of types obtainable from two primitive types, that of individuals (ι) and that of propositions ($\circ$) by repeated application of the function type construction. The function type is written by Church as juxtaposition of the two types from which it is formed codomain first, the

whole enclosed in brackets. Thus $(o\iota)$ is a type of propositional functions, functions whose domain is the individuals and whose codomain is the propositions. These may be thought of as first-order predicates.

We will work henceforth with the type system as is appears in ProofPower-HOL. The differences in the type systems insofar as they bear upon the present section are, firstly that the type of individuals is called *IND* rather than ι and secondly that the instead of a type of *propositions* ProofPower has a type of *truth values*, *BOOL*.[1] For illustrative purposes we work as if the primitive types were the natural numbers, $\mathbb{N}$ and the truth values *BOOL*.

Types will normally be shown in "Quine corners" thus: $\ulcorner:\mathbb{N}\urcorner$ is the type of natural numbers and $\ulcorner$:BOOL$\urcorner$ the type of truth values. We work with a "meta-language" which is a functional programming language, and which allows us to manipulate syntax using a computer, and provides facilities for constructing and checking formal proofs. The text will be interspersed with short scripts which should be understood as being submitted to an interactive proof tool in sequence.

For example:

SML

```
val one = 1;
```

gives the name *one* to the number 1. We may also sometimes show the response of the tool to such an input. Thus, to the input:

SML

```
one+one;
```

the system will now respond:

ProofPower output

```
val it = 2 : int
```

showing that ProofPower has evaluated the expression *one + one* and concluded that its value is the integer 2.

Standard ML (SML) is itself closely related to the typed lambda calculus which is our subject matter, so we have similar type systems and facilities for functional abstraction both in our metalanguage and in our object language. In order to talk about the object language type system in our metalanguage we have in SML a type *TYPE* which has a structure corresponding to that of the types of the object language. When an expression is entered in the type quote brackets $\ulcorner:$ and $\urcorner$ it is construed as an object of type TYPE representing a type of the object language, and can then be manipulated as such.

The exchange:

SML

```
⌜:ℕ⌝;
```

ProofPower output

```
val it = ⌜:ℕ⌝ : TYPE
```

shows that ProofPower has recognised the type we entered as a type.

[1] In Church the axiom of excluded middle is left optional, so that the logic need not be 'classical'. In ProofPower-HOL the axiom of exluded middle is stated as the assertion that there are just two truth values, T and F.

Function types may be written in the object language using the infix function type constructor $\rightarrow$, or may be computed using the metalanguage function $mk\_ \rightarrow \_type$. Thus a truth-valued function may have type:

SML
```
⌜:ℕ → BOOL⌝;
```

which may also be obtained by the following computation:

SML
```
mk_→_type (⌜:ℕ⌝, ⌜:BOOL⌝);
```

ProofPower Output
```
val it = ⌜:ℕ → BOOL⌝ : TYPE
```

Thus the heirarchy of types is the set of types which are obtainable from $\ulcorner : \mathbb{N} \urcorner$ and $\ulcorner : BOOL \urcorner$ by use of the metalanguage function $mk\_ \rightarrow \_type$ or the object language infix type constructor $\rightarrow$.

4 The Simply Typed Lambda Calculus

The simple types are introduced to serve as a type system for a typed $lambda - calculus$.

The lambda-calculus is the simplest imaginable notation for functional abstraction and application.

Functional abstraction is the process of obtaining a function from some expression containing variable symbols.

A function is a mapping from argument values to result values. When a function is to be defined using an expression, such as $\ulcorner x + x * x \urcorner$ the idea is that the value of the function for some argument is obtained by substituting the value of the argument for the variable in the expression and the resulting expression is then evaluated to give the value of the function for that argument. However, we want to be able to do this with expressions which contain more than one variable, and we then need to be able to stipulate which variable is to be used for the argument value.

Typically this might be done as follows:

```
f(x) = x + x*x
```

5 Church's Simple Theory of Types

SML
```
infix →;
datatype Type = ∘ | i | op → of Type * Type;
```

SML
```
fun   print_Type ∘ = "∘"
|     print_Type i = "i"
|     print_Type (f → a) = concat [if f = ∘ orelse f = i then print_Type f else concat ["(", print_Type f
;
```

```
SML
infix ^;
fun a ^ 0 = a |
    a ^ n =       let val a' = a ^ (n - 1)
                  in (a' -> a') -> (a' -> a')
                  end;

infix a;
datatype Term = V of string * Type
              | C of string * Type
              | op a of Term * Term
              | λ of string * Type * Term;
```

```
SML
fun   atomic_Term (V (s, t)) = true
|     atomic_Term (C (s, t)) = true
|     atomic_Term x = false;

fun print_Term (V (s, t)) = concat["(", s, ":", print_Type t, ")"]
|     print_Term (C (s, t)) = concat["(", s, "::", print_Type t, ")"]
|     print_Term (f a a) = concat[
              print_Term f,
              if atomic_Term a then print_Term a else concat["(", print_Term a, ")"]]
|     print_Term (λ (s, ty, tm)) = concat ["λ", s, ":", print_Type ty, print_Term tm];
```

```
SML
infix 8 a;
```

```
SML
exception Ill_typed_term of string;

fun typ_of (V(s, t)) = t
|   typ_of (C(s, t)) = t
|   typ_of (λ (s,ty,te)) = ty -> (typ_of te)
|   typ_of (f a a) = let val (tfd -> tfc) = typ_of f
                         and ta = typ_of a
                         in      if tfd = ta
                           then tfc
                           else raise Ill_typed_term (translate_for_output ((concat[
                                 "function:", print_Term f, " :", print_Type (tfd -> tfc), " ",
                                 "argument:", print_Term a, " :", print_Type (ta)])))
                         end;

typ_of ((C ("a", i -> ∘)) a (V ("a", i)));
```

SML

```
fun ~ A = C("N", (o -> o)) a A;
infix |;
fun A | B = C("A", o -> o -> o) a A a B;
infix &;
fun A & B = ~(~ A | ~ B);
infix >;
fun A > B = (~ A | B);
infix =;
fun A = B = (A > B) & (B > A);
fun ! A = C("||-", ((typ_of A) -> o)) a A;
fun ? A = ~ (! (~ A));
fun i A = let val ta = typ_of A
      in C("i", ta -> o) a A
      end;
infix Q;
fun A Q B =
      let   val ft = (typ_of A) -> o;
            val f = V("f", ft)
      in ! (L("f", ft, f a A > f a B))
      end;
infix #;
fun A # B = ~ (A Q B);
fun I_t a = L("x", a, V ("x", a));
fun I A = (I_t (typ_of A)) a A;
fun K_t a1 a2 = L("x", a1, L("y", a2, V ("x", a1)));
fun K A B = K_t (typ_of A) (typ_of B) a A a B;
fun Z_t a = L("f", a -> a, I_t a);
fun Nat_t a n = L("f", a -> a, L("x", a, fun_pow n (fn t => V("f", a -> a) a t) (V("x", a))));
fun Suc_t a =
      let val f = V("f", a -> a);
          val n = V("n", a ^ 1);
          val x = V("x", a)
      in L("n", a ^ 1, L("f", a -> a, L("x", a, (f a ((n a f) a x)))))
      end;
fun Suc A = (Suc_t (typ_of A)) a A;
fun N_t a = L("n", a ^ 1,
      ! (L("f", ((a ^ 1) -> o),
        (V ("f", (a ^ 1) -> o) a (Z_t a))
        > (!       (L("x", a ^ 1,
              (V("f", (a ^ 1) -> o) a (V("x", a ^ 1)))
              > (V("f", (a ^ 1) -> o) a Suc(V("x", a ^ 1)))
```

```
|            ) )
|        ⊃ (V("f", (a ⌢ 1) → ∘) a (V("n", a ⌢ 1))
|          )
|      ))));
```

6 HOL

6.1 Origins

Formal Higher Order Logics originate in Russell's Theory of Types [4] which is a so-called *ramified* type theory, the ramifications in which are obviated by the adoption of an axiom of reducibility. That the effect could be obtained more economically by omitting both the ramifications and the axiom of reducibility was quickly noted, and the resulting Simple Theory of Types was first codified in detail by Rudolf Carnap in his *Abriss Der Logistik*[1].

Alonzo Church, for the purpose of answering the 'enscheidungsproblem' distilled the essence of functional abstraction into the lambda-calculus, and subsequently investigated whether this calculus could be made the basis for a logical systems strong enough for mathematics. An attempt using a type-free lambda-calculus having foundered, Church put forward an elegant and economic formulation of the Simple Theory of Types [2] based on a simply-typed lambda-calculus. Later, Robin Milner (a computer scientist), for purposes connected with program verification using a logic for computable functions devised by Dana Scott (a set theorist), devised a polymorphic version of the simply typed lambda calculus (i.e. one in which type variables were permitted), and lead the development of LCF, an interactive proof assistant in which formal proofs in LCF could be constructed and checked using a functional programming language also benefitting from a polymorphic type system.

The version of Higher Order Logic which is used in this document was then devised by Mike Gordon for the formal verification of digital hardware. It consists of Church's formulation of the Simple Theory of Types made polymorphic, augmented by principles of convervative extension and implemented following the LCF paradigm as a proof assistant. The present document was prepared using a subsequently developed proof assistant for that same logical system, again following the patterm set by LCF.

6.2 The HOL Language

For a fully detailed account of the ProofPower-HOL language see the ProofPower Description Manual[5]. Tutorial material may be found in [7, 6]. The following is a very simplified and condensed account of the language which might possibly be sufficient for readers of this document.

6.2.1 Types

The types are defined inductively as follows.

A type may be any of the following:

- a type-variable
- a type construction consisting of a type constructor applied to a (possibly empty) finite sequence of types

6.2.2 Terms

Typed lambda terms consist of terms of the lambda-calculus annotated by types in the above type system.

The terms are defined inductively as follows.

A term may be any of the following:

1. a variable consisting of a name and a type
2. a constant also consisting of a name and a type
3. an application consisting of two terms, a function on the left and an argument on the right
4. an abstraction consisting of a variable (name and type) and a body (a term).

Terms must be well-typed, and if well-typed will have a most general polymorphic simple type. The following rules must be observed.

In an application the function must have a type of the form $\ulcorner : \alpha \to \beta \urcorner$ and the argument must have type $\ulcorner : \alpha \urcorner$. The application will then have type $\ulcorner : \beta \urcorner$

The type of an abstraction is formed from the type of the variable and the type of the body, and is the type of functions from the first to the second. It is written as a lambda abstraction.

If b has type $\ulcorner : \beta \urcorner$ then the lambda abstraction $\ulcorner \lambda x : \alpha \bullet b \urcorner$ will have type $\ulcorner : \alpha \to \beta \urcorner$.

In the concrete syntax of terms the presentation of applications is controlled by the use of fixity declarations which indicate whether the application of a variable or constant of a give name should be written prefix (with the function on the left of the argument), postfix (with the function on the right), or infix (with the function bewteen two arguments) and also permits a precedence to be assigned which controls how the parser inserts omitted brackets. Functions take only one argument which need not be enclosed in brackets, however, that argument might be a pair (or a pair of pairs) formed by the infix operator $\ulcorner \$, \urcorner$, in which case it would normally be enclosed in brackets. As in this example, the lexical status of a function may be suspended by preceding it with a dollar sign, and this is necessary if the function is to be used as an argument to some other function.

Type variables begin with a backquote ‘. Variable and constant names may include special symbols.

6.2.3 Formulae and Sequents

A formula is a term of type $\ulcorner$:BOOL$\urcorner$.

The logical system is a Hilbert style sequent calculus. A sequent is a list of formulae (the assumptions) together with a single formula (the conclusion). A sequent which has not been proven is called a goal or a conjecture. Sequents which have been proven have a special ML type THM. Objects of this type can only be computed by computations which parallel the structure of the logical system with the effect that all objects of type THM are indeed theorems of the logic derivable in an appropriate context. The relevant kind of context is call a theory, and determines what definitions and/or non-logical axioms are in scope. Theories are organised into a heirarchy, each theory except ”min” having at least one parent and inheriting all definitions and axioms from its ancestors. Proven theorems may be saved in a theory, in which case they will be listed with the theory.

6.2.4 Primitive Type Constructors and Constants

The primitive type constructors and constants are introduced in theory min.

There are three primitive type constructors. Two 0-ary constructors (type constants), a type of individuals $\ulcorner : IND \urcorner$ and a type of truth values $\ulcorner : BOOL \urcorner$, and a 2-ary constructor $\rightarrow$ (function space) normally written infix ($\ulcorner : BOOL \rightarrow BOOL \urcorner$ is the type of unary functions over type $\ulcorner : BOOL \urcorner$).

There are just three primitive constants:

$\Rightarrow$	$: BOOL \rightarrow BOOL \rightarrow BOOL$	material implication
$=$	$:' a \rightarrow 'a \rightarrow BOOL$	equality
ϵ	$: ('a \rightarrow BOOL) \rightarrow 'a$	the choice function

6.2.5 Primitive Definitions

Certain definitions are required before the axioms of HOL can be stated. These define the logical connectives and quantifiers and the concepts of injection and surjection which are needed in the axiom of infinity and in defining new types.

T	$: BOOL$
$\forall$	$: ('a \rightarrow BOOL) \rightarrow BOOL$
$\exists$	$: ('a \rightarrow BOOL) \rightarrow BOOL$
F	$: BOOL$
$\neg$	$: BOOL \rightarrow BOOL$
$\wedge$	$: BOOL \rightarrow BOOL \rightarrow BOOL$
$\vee$	$: BOOL \rightarrow BOOL \rightarrow BOOL$
OneOne	$: ('a \rightarrow 'b) \rightarrow BOOL$
Onto	$: ('a \rightarrow 'b) \rightarrow BOOL$
TypeDefn	$: ('b \rightarrow BOOL) \rightarrow ('a \rightarrow 'b) \rightarrow BOOL$

These definitions may be found in theory log.

6.2.6 Fixity and Sugar

In order to make HOL terms more readable, certain special syntactic forms are accepted by the parser which are closer to normal mathenatical notation than would otherwise be acceptable. Full details of the accepted concrete syntax are shown in the ProofPower Description Manual [5]

	Sugared	**Unsweetened**
binders	$\forall x \bullet x + x$	$\$\forall\ (\lambda x \bullet x + x)$
paired abstractions	$\lambda(x, y) \bullet x$	$Uncurry\ (\lambda x\ y \bullet x)$
let clauses	$let\ a = 4\ in\ a * a$	$Let\ (\lambda a \bullet a * a)\ 4$
if clauses	$if\ a = 4\ then\ 0\ else\ a * a$	$Cond\ (a = 4)\ 0\ (a * a)$
set displays	$\{1; 2; 3\}$	$Insert\ 1\ (Insert\ 2\ (Insert\ 3\ \{\}))$
set comprehension	$\{x \mid x \leq 34\}$	$SetComp\ (\lambda x \bullet x \leq 34)$
list displays	$[1; 2; 3]$	$Cons\ 1\ (Cons\ 2\ (Cons\ 3\ []))$

6.3 The HOL Deductive System

6.3.1 Axioms

There are five axioms.

$bool\_cases\_axiom$	asserts that T and F are the only values of type BOOL.
$\Rightarrow\_antisym\_axiom$	asserts that implication is antisymmetric.
η_axiom	makes functions extensional.
ϵ_axiom	is the axiom of choice.
$infinity\_axiom$	is the axiom of infinity.

These axioms may be found in theory init.

6.3.2 Rules of Inference

There are six primitive rules of inference in Church's STT:

I Alpha conversion.

II Beta contraction.

III Beta expansion.

IV Substitution for free variables

V Modus Ponens

VI Universal Generalisation

Our system also needs a rule permitting the instantiation of type variables.

6.3.3 Definitions and Other Extensions

There are three kinds of extension which are possible. Axioms, definitions and conservative extensions.

An axiom is an assertion of a sequent without proof. These are always stored as such in the current theory when they are asserted so that the axioms used in deriving theorems can be determined. HOL is a powerful logical system sufficient for the development of a large part of mathematics using only definitions or conservative extensions over the primitive logical axioms. It is therefore rarely necessary to adopt new axioms, and it is the norm to proceed by definitions and conservative extensions.

7 ProofPower

8 Conclusions

9 Postscript

My initial ideas about the content of the document are now apparent in the content list.

I'm not yet sure how it will work out, trying to use ProofPower to illuminate the account, when more is in place I may have more to say about the future.

A Theory Listings

A.1 The Theory grice

Parents

rbjmisc

Children

griceS *griceC*

Constants

$\$\models$	$BOOL\ LIST \to BOOL \to BOOL$
$\$\boldsymbol{committal}$	$('a \to BOOL) \to ('a \to BOOL) \to BOOL$

Fixity

Right Infix 4: $\models$
Right Infix 100:
committal

Definitions

$\models$	$\vdash \forall\ al\ c\bullet\ (al \models c) \Leftrightarrow \forall_L\ al \Rightarrow c$
committal	$\vdash \forall\ P\ \phi\bullet\ P\ committal\ \phi \Leftrightarrow (\forall\ \alpha\bullet\ \phi\ \alpha \Rightarrow P\ \alpha)$

Theorems

ASS	$\vdash \forall\ \phi\bullet\ [\phi] \models \phi$
RAA	$\vdash \forall\ \phi\ \psi\ \zeta\ \Gamma\bullet\ ([\phi]\ @\ \Gamma \models \psi \wedge \neg\ \psi) \Rightarrow (\Gamma \models \neg\ \phi)$
DN	$\vdash \forall\ \phi\bullet\ [\neg\ \neg\ \phi] \models \phi$
$\wedge\boldsymbol{I}$	$\vdash \forall\ \phi\ \psi\bullet\ [\phi;\ \psi] \models \phi \wedge \psi$
$\wedge\boldsymbol{E}$	$\vdash \forall\ \phi\ \psi\bullet\ ([\phi \wedge \psi] \models \phi) \wedge ([\phi \wedge \psi] \models \psi)$
$\vee\boldsymbol{I}$	$\vdash \forall\ \phi\ \psi\bullet\ ([\phi] \models \phi \vee \psi) \wedge ([\phi] \models \psi \vee \phi)$
$\vee\boldsymbol{E}$	$\vdash \forall\ \phi\ \psi\ \zeta\ \Gamma\ \Delta\ \Theta$ $\bullet\ ([\phi]\ @\ \Gamma \models \zeta) \wedge ([\psi]\ @\ \Delta \models \zeta) \wedge (\Theta \models \phi \vee \psi)$ $\Rightarrow (\Gamma\ @\ \Delta\ @\ \Theta \models \zeta)$
CP	$\vdash \forall\ \phi\ \psi\ \Gamma\bullet\ ([\phi]\ @\ \Gamma \models \psi) \Rightarrow (\Gamma \models \phi \Rightarrow \psi)$
MPP	$\vdash \forall\ \phi\ \psi\bullet\ [\phi \Rightarrow \psi;\ \phi] \models \psi$
$\forall\boldsymbol{I}$	$\vdash \forall\ \Gamma\ P\bullet\ (\forall\ x\bullet\ \Gamma \models P\ x) \Rightarrow (\Gamma \models (\forall\ x\bullet\ P\ x))$
$\forall\boldsymbol{E}$	$\vdash \forall\ \Gamma\ P\ c\bullet\ (\Gamma \models (\forall\ x\bullet\ P\ x)) \Rightarrow (\Gamma \models P\ c)$
$\exists\boldsymbol{I}$	$\vdash \forall\ \Gamma\ P\ x\bullet\ (\Gamma \models P\ x) \Rightarrow (\Gamma \models (\exists\ x\bullet\ P\ x))$
$\exists\boldsymbol{E}$	$\vdash \forall\ P\ Q\bullet\ (\forall\ x\bullet\ [P\ x] \models Q) \Rightarrow ([\exists\ x\bullet\ P\ x] \models Q)$
P_2	$\vdash \forall\ P\ \phi\bullet\ P\ committal\ \phi \Rightarrow P\ committal\ (\lambda\ \alpha\bullet\ \neg\ \neg\ \phi\ \alpha)$
P_3	$\vdash \forall\ P\ \phi\ \psi$ $\bullet\ P\ committal\ \phi \vee P\ committal\ \psi$ $\Rightarrow P\ committal\ (\lambda\ \alpha\bullet\ \phi\ \alpha \wedge \psi\ \alpha)$
P_4	$\vdash \forall\ P\ \phi\ \psi$

$\bullet\ P\ committal\ \phi \wedge P\ committal\ \psi$
$\Rightarrow P\ committal\ (\lambda\ \alpha\bullet\ \phi\ \alpha \vee \psi\ \alpha)$

P_5 $\vdash \forall\ P\ \phi\ \psi$
$\bullet\ P\ committal\ (\lambda\ \alpha\bullet \neg\ \phi\ \alpha) \wedge P\ committal\ \psi$
$\Rightarrow P\ committal\ (\lambda\ \alpha\bullet\ \phi\ \alpha \Rightarrow \psi\ \alpha)$

P_6∀ $\vdash \forall\ P\ \phi$
$\bullet\ (\exists\ \beta\bullet\ P\ committal\ \phi\ \beta)$
$\Rightarrow P\ committal\ (\lambda\ \alpha\bullet \forall\ \beta\bullet\ \phi\ \beta\ \alpha)$

P_6∃*b* $\vdash \forall\ P\ \phi$
$\bullet\ (\forall\ \beta\bullet\ P\ committal\ \phi\ \beta)$
$\Rightarrow P\ committal\ (\lambda\ \alpha\bullet \exists\ \beta\bullet\ \phi\ \beta\ \alpha)$

A.2 *

Bibliography

[1] Rudolf Carnap. *Abriss der Logistik*. Verlag von Julius Springer. 1929.

[2] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–, 1940.

[3] Roger Bishop Jones. *Analyses of Analysis: Part I - Exegetical Analysis*. RBJones.com. 2009. http://www.rbjones.com/rbjpub/pp/doc/b001.pdf.

[4] B. Russell. Mathematical Logic as based on the Theory of Types. *American Journal of Mathematics*, 30:222–262, 1908.

[5] ds/fmu/ied/usr005. *ProofPower Description*. Lemma 1 Ltd., `http://www.lemma-one.com`.

[6] ds/fmu/ied/usr013. *ProofPower HOL Tutorial*. Lemma 1 Ltd., `http://www.lemma-one.com`.

[7] ds/fmu/ied/usr022. *ProofPower HOL Tutorial Transparencies*. Lemma 1 Ltd., `http://www.lemma-one.com`.

A.3 *

Index