

THE HENRY EDWIN SEVER GRADUATE SCHOOL

MASTER OF SCIENCE DEGREE

THESIS ACCEPTANCE

(To be the first page of each copy of the thesis)

DATE: April 24, 2006

STUDENT'S NAME: Li-Yang Tan

This student's thesis, entitled Formalizing the Meta-Theory of \mathcal{Q}_0 in the Calculus of Inductive Constructions has been examined by the undersigned committee of three faculty members and has received full approval for acceptance in partial fulfillment of the requirements for the degree Master of Science.

APPROVAL: _____ Chairman

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

FORMALIZING THE META-THEORY OF \mathcal{Q}_0 IN THE CALCULUS OF
INDUCTIVE CONSTRUCTIONS

by

Li-Yang Tan

Prepared under the direction of Professor Aaron D. Stump

A thesis presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

May 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY
THE HENRY EDWIN SEVER GRADUATE SCHOOL
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

FORMALIZING THE META-THEORY OF \mathcal{Q}_0 IN THE CALCULUS OF
INDUCTIVE CONSTRUCTIONS

by Li-Yang Tan

ADVISOR: Professor Aaron D. Stump

May 2006

Saint Louis, Missouri

The notion of a proof is central to all of mathematics. In the language of formal logic, a proof is a finite sequence of inferences from a set of axioms, and any statement one yields from such a finitistic procedure is called a theorem. For better or for worse, this is far from the form a traditional mathematical proof takes. Mathematicians write proofs that omit routine logical steps, and details deemed tangential to the central result are often elided. These proofs are fuzzy and human-centric, and a great amount of context is assumed on the part of the reader. While traditional proofs are not overly symbolic or syntactic, and hence are easily understood, such informal proofs are susceptible to logical errors – Fermat’s Last Theorem and the Four Color Theorem being prime examples. In light of this, there has been significant interest in producing formal proofs of mathematical theorems: proofs in which every intermediate logical step is supplied. Drawing on ideas from Computational Logic, Type Theory and the theory of Automated Deduction, we are able to guaranteed the

correctness of these proofs.

The formalization of mathematics is an endeavor that has enjoyed very encouraging progress in recent years. Major achievements include the complete formalization of the Four Color Theorem, the Prime Number Theorem, Goedel's Incompleteness Theorem, the Jordan Curve Theorem (all within the past five years!). This thesis presents our work in formalizing the meta-theory of Peter Andrews' classical higher-order logic \mathcal{Q}_0 in a higher-order typed lambda calculus. Our development is a completely formal one – in addition to formalizing \mathcal{Q}_0 's logical meta-theory, we have also developed and formalized the syntactic meta-theory of \mathcal{Q}_0 . Our syntactic meta-theory allows for the reasoning of notions such as variable occurrences, scope and variable binding, linear replacement, etc. Our formalization is carried out in the interactive proof assistant Coq, developed as part of the LogiCal Project in INRIA. Coq is built upon the Calculus of Inductive Constructions, an extension of Coquand and Huet's seminal Calculus of Construction with support for inductive data types. As far as we know, this thesis presents the first effort to formalize Andrews' logical system.

Contents

Acknowledgments	v
1 Introduction	1
1.1 The Formalization of Mathematics	1
1.1.1 Proofs and Truth	1
1.1.2 The Importance of Being Formal	2
1.2 The Meta-Theory of \mathcal{Q}_0 in Coq	3
2 Type Theory	5
2.1 Lambda Calculus	5
2.1.1 Untyped Lambda Calculus	6
2.1.2 Typed Lambda Calculus	7
3 The Calculus of Inductive Constructions and Coq	11
3.1 Calculus of Constructions	11
3.2 The Interactive Proof Assistant Coq	11
4 \mathcal{Q}_0 in Coq	13
4.1 Peter Andrew’s Classical Higher Order Logic \mathcal{Q}_0	13
4.2 A Central Challenge: Formalizing the Syntactic Meta-Theory	14
4.2.1 Previous work: The Sublogic in RSP	15
4.3 The Primitive Basis of \mathcal{Q}_0	15
4.3.1 \mathcal{Q}_0 types, constants and terms	15
4.3.2 Definitions of Logical Operators in \mathcal{Q}_0	19
4.3.3 Axioms and Inference Rules	20
4.4 The Syntactic Meta-Theory	23
4.5 The Meta-Theorems of \mathcal{Q}_0	26
5 Conclusion	28
Appendix: Coq Source Code	30

References	63
Vita	65

Acknowledgments

First, I am grateful to Aaron Stump for guiding me expertly through this thesis work and several related projects. My time in the Computational Logic Group has been the most enjoyable, and I thank Aaron for his leadership of our group. Next, I am deeply indebted to Sally Goldman for her advice and tutelage over the years – my interactions with Sally have gone a long way in shaping my academic aspirations. It has been a pleasure working with and learning from her in various capacities. I also thank John Shareshian for his superb supervision of my work in discrete mathematics, and for the many enjoyable conversations about the Green-Tao Theorem, New York City, and everything in between. As I go off to graduate school, I can only hope to be guided along by advisors like Aaron, Sally and John.

I thank Jeremy Buhler, Sally Goldman, András Gyárfás, Ervin Györi, Peter Komjáth, Aaron Stump and Nik Weaver for their inspiring instruction in some of my favorite classes. In particular, I thank Jeremy and Sally for serving on my thesis committee. I also thank Morgan Deters, Ian Wehrman, Eddy Westbrook and Bob Zimmermann for many helpful discussions about this work.

Finally, I am grateful to Chancellor Mark Wrighton for his enthusiastic support of my academic endeavors. Getting to know Chancellor Wrighton this past year has been a great honor and pleasure.

Li-Yang Tan

Washington University in Saint Louis
May 2006

Chapter 1

Introduction

1.1 The Formalization of Mathematics

1.1.1 Proofs and Truth

I was interviewed in the Israeli Radio for five minutes and I said that more than 2000 years ago, Euclid proved that there are infinitely many primes. Immediately the host interrupted me and asked: “Are there still infinitely many primes?”

Noga Alon

The centrality of the notion of a *proof* in mathematics cannot be overstated. In the language of formal logic, a proof is a finite sequence of inferences from a set of axioms. Any statement one yields from this finitistic procedure is called a *theorem*, and theorems are acknowledged as unequivocal mathematical truth. The finality and permanence of a proof is what distinguishes mathematics from all other scientific pursuits – theorems are irrefutable once established. Euclid proved in 300 BC that for any finite set $\{p_1, \dots, p_k\}$ of primes, there exists a prime p greater than all of them. The infinitude of primes still holds as mathematical truth today, and the same will be true tomorrow. Such is the importance of proofs that an entire branch of meta-mathematics and philosophy, known as Proof Theory, is devoted precisely to the study of proof and mathematical truth. Proof theorists treat proofs as mathematical objects, and reason syntactically about the power of proof and logical systems.

Readers acquainted with theoretical computer science will know that the notion of a proof is also the basis of computational complexity theory [10]. In complexity theory, one is concerned with how efficiently a proof can be found mechanically, and further, if the validity of proofs can be checked quickly; efficiency in both cases is measured in terms of time and space. We recall that the complexity class P is precisely the collection of all decision problems whose proofs can be found in polynomial time, and NP is the collection of all decision problems with polynomial-sized proofs. The P versus NP question, which asks if the class P is equal in power to NP , has distinguished itself as the central question in theoretical computer science for nearly four decades now.

1.1.2 The Importance of Being Formal

What then, is a *formal* proof? As mentioned in the previous section, a proof is a finite sequence of logic inferences, such as modus ponens, starting from a set of axioms, typically Zermelo-Frankel with Choice (ZFC). This is akin to the how elementary facts in plane geometry are established in high-school mathematics classes. However, this is in fact far from the form a mathematical proof takes in practice. Professional mathematicians write proofs that are fuzzy and human-centric. Pedantic details deemed tangential to the core of the result are often conveniently elided. To further underscore the distinction between formal proofs and proofs mathematicians write in practice, we quote Thomas Hales [9].

Traditional mathematical proofs are written in a way to make them easily understood by mathematicians. Routine logical steps are omitted. An enormous amount of context is assumed on the part of the reader . . . In a formal proof, all the intermediate logical steps are supplied. No appeal is made to intuition, even if the translation from intuition to logic is routine. Thus, a formal proof is less intuitive, and yet less susceptible to logical errors.

This is the notion of a formal proof we will adopt from here on out — a formal proof is one in which *all* logical steps are provided.

We bring the readers' attention to the last sentence of the above quote, where Hales claims that “a formal proof is less intuitive, and yet *less susceptible to logical*

errors.” Setting aside for a moment doubt about the validity of such an assertion, we point out that herein lies the motivation behind all work in the formalization of mathematics: Formal proofs provide us with greater, if not total, confidence that a purported theorem is indeed true.

The field of mathematics is certainly no stranger to alleged proofs of theorems that were later found to contain logical errors. Perhaps the most famous incident in recent years is Andrew Wiles’ proof of Fermat Last Theorem. After announcing his proof in a dramatic series of three lectures at Cambridge University in June of 1993, Wiles received world-wide recognition for having resolved what was undoubtedly the biggest open question of our times, left open from the 17th century. Unfortunately, however, a serious technical error in Wiles’ proof was uncovered not long after. Wiles consequently spent a year fixing his proof and, with the help of his former student Richard Taylor, Fermat’s Last Theorem was finally put to rest in September of 1994. Readers familiar with the history of the Four Color Theorem will know that prior to Appel and Haken’s 1976 proof, the question was twice assumed to have been settled. Both claims turned out to be based on erroneous proofs — the first is due to Alfred Kempe in 1879, and the second, Peter Tait in 1880. In both cases, the false proofs were universally accepted by the mathematical community and stood unchallenged for *over a decade* before a counter-example was found!

1.2 The Meta-Theory of \mathcal{Q}_0 in Coq

This thesis presents our contributions to the aforementioned efforts in the formalization of mathematics. We present our work in formalizing the meta-theory of Peter Andrews’ classical higher-order logic \mathcal{Q}_0 in the Calculus of Inductive Constructions. \mathcal{Q}_0 is a minimal classical logic based on untyped lambda calculus, with all its theory built on just five axioms and one inference rule. Our formalization is carried out in the interactive proof assistant Coq, developed as part of the LogiCal Project in INRIA. Coq is built upon the Calculus of Inductive Constructions, an extension of Coquand and Huet’s seminal Calculus of Construction with support for inductive data types.

Our first contribution is a complete formalization of the primitive basis of \mathcal{Q}_0 . The core of \mathcal{Q}_0 consists of \mathcal{Q}_0 types, variables, constants and terms, definitions of logical operators and syntactic abbreviations, its five axioms and the rule of inference.

\mathcal{Q}_0 's rule of inference, Rule R, is a rule for linear replacement of terms that allows for variable capture.

The main contribution of this thesis work is the development and formalization of \mathcal{Q}_0 's syntactic meta-theory. While Peter Andrews is impressively meticulous in his presentation of logical derivations in the meta-proofs of \mathcal{Q}_0 , many syntactic derivations that are arguable intuitive have been elided. However, in a completely formal development such as ours, all derivations, be they logical or syntactic, have to be made explicit in the formalization. Hence, we have developed and formalized in Coq a meta-theory of \mathcal{Q}_0 's syntactic notions, including variable occurrences, binding and scope, linear and full replacement etc. Further, we have implemented tactics and tacticals that automate the process of searching and building proofs syntactic properties. We are able to guarantee soundness and completeness for our tactics and tacticals.

Last, having formalized the syntactic meta-theory of \mathcal{Q}_0 in Coq, we have since gone on to provide formal proofs of some of the elementary meta-theorems of \mathcal{Q}_0 . This includes basic properties of equality in \mathcal{Q}_0 , such as reflexivity, symmetry and transitivity, along with restricted and generalized versions of beta- and eta-reduction in \mathcal{Q}_0 . It is our hope that this project will culminate in the proving of the Deduction Theorem of \mathcal{Q}_0 . To the best of our knowledge, this is the first successful effort to formalize the meta-theory of Peter Andrews' logic.

Chapter 2

Type Theory

An effectively calculable function of the positive integers is a λ -definable function of the positive integers.

Church's Thesis

2.1 Lambda Calculus

The λ -calculus is a formal mathematical system developed to formalize and reason about the notions of computation and computability. The *Entscheidungsproblem*, German for “decision problem”, is an ideal set forth by Gottfried Leibniz in the seventeenth century centered around a philosophical question which asked if there is a way to solve all problems formulated in a universal language, such as the language of set theory and first order predicate logic. This question was again posed by David Hilbert in 1928 in continuation of his program initiated at the turn of the century, and is therefore also commonly known as *Hilbert's Entscheidungsproblem*.

The *Entscheidungsproblem* was answered in the negative in 1936 independently by Alonzo Church and Alan Turing. In his seminal paper “On computable numbers, with an application to the Entscheidungsproblem”, Turing introduced the construction of a universal computing machine, called a Turing Machine, with which he used to formalize the notion of computability. Church, in his papers “An unsolvable problem of elementary number theory” and “A note on the Entscheidungsproblem”, invented the λ -calculus and studied the notion of computable functions via this system. It is

a consequence of the celebrated Church-Turing thesis that these two approaches are in fact equivalent.

2.1.1 Untyped Lambda Calculus

Central to the aesthetic appeal of lambda calculus is its simplicity and elegance. The syntax of lambda calculus consists of just three sorts: *variables*, *function abstraction*, and *application*.

Formally, the set of all λ -terms, denoted Λ , is built up from a countably infinite set of variables $V = \{v_1, v_2, \dots\}$ using application and function abstraction. We define Λ inductively as follows.

$$\begin{aligned} x \in V &\Rightarrow x \in \Lambda \\ M, N \in \Lambda &\Rightarrow (MN) \in \Lambda \\ M \in \Lambda, x \in V &\Rightarrow \lambda x.M \in \Lambda \end{aligned}$$

The convention we adopt is for x, y, z, \dots to denote variables in V and for M, N, L, \dots to denote arbitrary λ -terms.

For a λ -term M , the set of *free variables* of M , denoted $FV(M)$, is defined inductively as follows

$$\begin{aligned} FV(x) &= \{x\} \\ FV(MN) &= FV(M) \cup FV(N) \\ FV(\lambda x.M) &= FV(M) - \{x\} \end{aligned}$$

If $x \in FV(M)$, we say that x is *free in* M . A variable in M that is not in $FV(M)$ is *bound in* M . M is called a *closed λ -term*, or *combinator* if $FV(M) = \emptyset$. The set of closed λ -terms is denoted by Λ^0 .

The result of *substituting* N for all free occurrences x in M , denoted $M[x \mapsto N]$, is defined inductively as follows.

$$\begin{aligned} x[x \mapsto N] &\equiv N \\ y[x \mapsto N] &\equiv y \quad \text{if } x \neq y \\ (M_1 M_2)[x \mapsto N] &\equiv (M_1[x \mapsto N])(M_2[x \mapsto N]) \\ (\lambda y. M_1)[x \mapsto N] &\equiv \lambda y. (M_1[x \mapsto N]) \end{aligned}$$

We would like to develop a formal notion of equivalence between λ -terms. Equivalence in the theory of λ -calculus is given by three axiom schemes. First, α -*conversion* rule formalizes the notion that renaming bound variables yield the same λ -term. For example, we would naturally want $\lambda x. x$ and $\lambda y. y$ to be considered the same term. Therefore, we have

α -equivalence: $\lambda x. M =_\alpha \lambda y. M[x \mapsto y]$, provided y does not occur in M .

Next, the β -*reduction* rule expresses the idea of function application. This is the principle axiom scheme of the theory of λ -calculus.

β -reduction: $(\lambda x. M)N \rightarrow_\beta M[x \mapsto N]$

Last, the η -*conversion* rule expresses the notion of *extensionality*. That is, two functions are considered the same if they agree on all input arguments.

η -conversion: $\lambda x. (Mx) \rightarrow_\eta M$, provided $x \notin FV(M)$.

2.1.2 Typed Lambda Calculus

Thus far, we have considered what is known as *type-free* λ -calculus, where every expression can be applied to every other expression, the former considered as a function and the latter, an argument. For example, the identity function $\lambda x. x$ may be applied to any λ -term to yield the term itself; in particular, $\lambda x. x$ may be applied to itself as follows

$$(\lambda x. x)(\lambda x. x) \rightarrow_\beta \lambda x. x$$

For an even more interesting example, consider the following λ -term

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

It is easy to see the above λ -term β -reduces to itself. Hence, we have the infinite chain of reductions

$$(\lambda x. x x)(\lambda x. x x) \rightarrow_{\beta} (\lambda x. x x)(\lambda x. x x) \rightarrow_{\beta} \dots$$

It is left to the reader to consider the result of applying the λ -term $(\lambda x. x (x x))$ to itself.

While undoubtedly an expressive and elegant system, we are often times interested in several properties that untyped λ -calculus does not possess. For example, we might want all terms to have a normal form. That is,

for all terms N , there exists a term \hat{N} such that $N \rightarrow_{\beta}^* \hat{N}$ and \hat{N} does not β -reduce to any other term.

Systems possessing such a property are called *normalizing*. It is easy to see from the examples above that untyped λ -calculus is certainly not normalizing. In particular, $(\lambda x.(x x))(\lambda x.(x x))$ does not have a normal form. For this reason and many others, we are keenly interested in *typed* versions of λ -calculus. We first consider the most basic, and in many ways canonical, form of typed λ -calculus known as *simply typed lambda calculus*. The only form of connective in simply typed lambda calculus is the function type \rightarrow . This system, denoted by λ^{\rightarrow} , was developed independently by Curry in 1934 and Church in 1940.

We will introduce the notion of a *typing relation* \mathcal{R} that associates with each λ -term in Λ its type. If M is a λ -term and a type σ is assigned to M , we say that ‘ M has type σ ’ and also that ‘ σ is inhabited by M .’ This judgment is typically denoted

$$M : \sigma$$

We will inductively defined the set of types \mathcal{T} of λ^{\rightarrow} . First, we have the collection \mathcal{B} of *type constants* for basic types such as **Nat**, the type of natural numbers, and **Bool**, the type of booleans. We also have a denumerable set of *type variables* $\mathcal{V} = \{\alpha, \alpha', \alpha'', \dots\}$. Instead of the somewhat cumbersome way of indexing α with apostrophes to differentiate between type variables, we often use lower-case Greek alphabets $\alpha, \beta, \gamma, \dots$ to denote arbitrary distinct types. \mathcal{T} is then defined inductively as follows.

$$\begin{aligned}
\alpha \in \mathcal{V} &\Rightarrow \alpha \in \mathcal{T} \\
B \in \mathcal{B} &\Rightarrow B \in \mathcal{T} \\
\sigma, \tau \in \mathcal{T} &\Rightarrow (\sigma \rightarrow \tau) \in \mathcal{T}
\end{aligned}$$

We often adopt the following more concise syntax, easily seen to be equivalent to the inductive definition above.

$$\begin{aligned}
\mathcal{T} &= \mathcal{V} \mid \mathcal{B} \mid \mathcal{T} \rightarrow \mathcal{T} \\
\mathcal{V} &= \alpha \mid \mathcal{V}
\end{aligned}$$

Finally, a *context* is a set of typing assumptions from which new type assignments can be derived. Contexts are typically denoted by Γ and often also Δ . If the type assignment $M : \sigma$ is derived from the context Γ , we say that ‘ $M : \sigma$ is derivable from Γ ’ and denote this judgment as

$$\Gamma \vdash M : \sigma$$

If $M : \sigma$ is derived from an empty set of assumptions, we write $\vdash M : \sigma$ as a shorthand for $\emptyset \vdash M : \sigma$.

We may now define the typing relation $\mathcal{R} \subset \Lambda \times \mathcal{T}$ of λ^\rightarrow by giving the axioms and inference rules for type derivations in this system. Notice that they correspond exactly to the three production rules in our inductive definition of Λ .

$$\begin{aligned}
&\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \text{ var} \\
&\frac{\Gamma \cup \{x : \sigma\} \vdash M : \tau}{\Gamma \vdash \lambda x : \sigma. M : \sigma \rightarrow \tau} \text{ abs} \\
&\frac{\Gamma \vdash M : \sigma \rightarrow \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash M N : \tau} \text{ app}
\end{aligned}$$

For example, we have

$$\begin{aligned}
&\vdash \lambda x : \alpha. x : \alpha \rightarrow \alpha \\
&\vdash \lambda x : \alpha. \lambda y : \beta. x : \alpha \rightarrow (\beta \rightarrow \alpha)
\end{aligned}$$

$\vdash \lambda x : \alpha \rightarrow \beta \rightarrow \gamma. \lambda y : \alpha \rightarrow \beta. \lambda z : \alpha. x z (y z) : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$

We note that the λ -term $(\lambda x. x x)$ is *untypable* in λ^\rightarrow . That is, $((\lambda x. x x), \alpha) \notin \mathcal{R}$ for all $\alpha \in \mathcal{T}$. In fact λ^\rightarrow is strongly normalizing, and consequently, $\beta\eta$ -equivalence is decidable. Here we state without proof six important properties of λ^\rightarrow .

Uniqueness of Types

If $\Gamma \vdash M : \sigma$ and $\Gamma \vdash M : \tau$, then $\sigma = \tau$.

Subject Reduction

If $\Gamma \vdash M : \sigma$ and $M \rightarrow_{\beta\eta} N$, then $\Gamma \vdash N : \sigma$.

Strong Normalization

If $\Gamma \vdash M : \sigma$, then all $\beta\eta$ -reductions from M terminate.

Substitution Property

If $\Gamma \cup \Delta \cup \{x : \tau\} \vdash M : \sigma$ and $\Gamma \vdash N : \tau$, then $\Gamma \cup \Delta \vdash M[x \rightarrow N] : \sigma$.

Weakening

If $\Gamma \vdash M : \sigma$ and $\Gamma \subset \Delta$, then $\Delta \vdash M : \sigma$.

Strengthening

If $\Gamma \cup \{x : \tau\} \vdash M : \sigma$ and $x \notin FV(M)$, then $\Gamma \vdash M : \sigma$.

Given any type system, several natural questions may be asked. In particular, we would like algorithms for the following problems.

1. Given a context Γ , a term M and a type σ , is it true that $\Gamma \vdash M : \sigma$?
2. Given a context Γ and a term M , does there exist a type σ such that $\Gamma \vdash M : \sigma$?
3. Given a type σ , does there exist a term M such that $\vdash M : \sigma$?

The above problems are known as the *Type Checking*, *Type Synthesis* and *Type Inhabitation* problems respectively. They are typically denoted as

$\Gamma \vdash M : \sigma?$	Type Checking Problem (TCP)
$\Gamma \vdash M : ?$	Type Synthesis Problem (TSP)
$\Gamma \vdash ? : \sigma$	Type Inhabitation Problem (TIP)

We note that TCP, TSP and TIP are all decidable for λ^\rightarrow , and TSP is in fact equivalent to TCP. We note also that TIP is typically undecidable even in the most modest extensions of λ^\rightarrow . For example, TIP is undecidable in Girard and Reynold's polymorphic lambda calculus System F.

Chapter 3

The Calculus of Inductive Constructions and Coq

3.1 Calculus of Constructions

The Calculus of Constructions (CoC) [7] is a higher-order typed lambda-calculus, developed jointly by Thierry Coquand and Gerard Huet in 1986. The CoC is intended as a higher-order formalism for constructive proofs in natural deduction style. In the spirit of the *Curry-Howard correspondence* as advocated by Martin-Löf Type Theory, every proof is a λ -expression in the Calculus of Constructions, typed with propositions of the underlying logic. In fact, CoC can be seen as an extension of the the Curry-Howard correspondence. The Curry-Howard correspondence establishes an isomorphism between terms in simply typed lambda calculus with natural-deduction style proofs in intuitionistic propositional logic; CoC extends this isomorphism to proofs in *full intuitionistic predicate calculus*.

3.2 The Interactive Proof Assistant Coq

Coq is an interactive proof assistant for the development of formal machine-checked proofs [5, 6, 14]. Developed as part of the LogiCal Project based in INRIA in France, it is written in the Objective Caml programming language. The Calculus of Inductive Constructions, an extension of CoC with inductive datatypes, is underlying theory that the Coq proof assistant works within.

Coq has made outstanding contributions to two broad fields within just two decades of its inception. First, Coq has been used extensively and effectively by mathematicians, logicians and computer scientists alike to produce formal proofs of mathematical theorems. Theorems that have been successfully formalized in Coq range from the simple and straightforward, such as the irrationality of $\sqrt{2}$, to the deep and profound, such as the Four Color Theorem, Gödel’s Incompleteness Theorem, and the Fundamental Theorem of Algebra. Beyond its contributions to the field of formalized mathematics, Coq has also been of great interest to software engineers interested in applying formal methods to the development of provably correct code and zero-fault software. To this end, Coq is used to first formally state software specifications, and then prove that corresponding code does indeed meet the desired specifications. Admittedly, it can be argued that these two fields are in fact unified – or rather, the second can be seen to be subsumed by the first – by the Curry-Howard correspondence.

Our work falls into the former category. It should be noted, though, that our project differs significantly in spirit to almost all previous work in the area. Most projects have been concerned with the formalization of mathematical theorems in Coq. Just as in mainstream mathematical practice, such formalizations implicitly assume an underlying logic and axiomatic system and work *within* the logical system. In this thesis, however, we are interested not in producing formal proofs of theorems, but instead in verifying the correctness of a particular logical system – our work is thus driven by foundational interests. In particular, our goal is to formalize the syntactic and logical meta-theory of \mathcal{Q}_0 , and to produce formal proofs of *meta-theorems* such as alpha-equivalence, beta-reduction and the deduction theorem. To draw an analogy, we are interested in proving properties about a programming language, instead of the correctness of code written in a programming language. In fact, the spirit of our project is very much akin to work done in Twelf [11, 12], an implementation of the logical framework LF is used extensively for the verification of programming language meta-theory.

Chapter 4

\mathcal{Q}_0 in Coq

*Science is what we understand well enough to explain to a computer,
Art is all the rest.*

Don Knuth

4.1 Peter Andrew's Classical Higher Order Logic \mathcal{Q}_0

Introduced by Peter Andrews in the 1960's [2, 3], \mathcal{Q}_0 is a classical higher-order logic based on simply-typed lambda calculus. There are several reasons why we are interested in formalizing the meta-theory of \mathcal{Q}_0 .

- A formal proof of \mathcal{Q}_0 's correctness would be of significant foundational interest. Peter Andrews uses \mathcal{Q}_0 as a formal framework to develop core areas of the foundations of mathematics, including cardinals and the axiom of infinity, Peano's postulates, primitive recursive functions, etc.
- \mathcal{Q}_0 is an elegant logical system with significant aesthetic appeal. It is a minimal logic based only on equality, with all its theory built on just 5 axiom schema and 1 inference rule.
- The complex syntax of \mathcal{Q}_0 makes it impossible for its correctness to be checked by hand. Consider, for example, the following definition of the \wedge operator

$$[\lambda x_o \lambda y_o \cdot [\lambda g_{ooo} \cdot g_{ooo} T T]] = [\lambda g_{ooo} \cdot g_{ooo} x_o y_o]$$

Peculiar syntactic conventions such as linear replacement allowing for variable capture pose additional challenges.

- Issues such as variable binding and replacement which we have to deal with explicitly are of contemporary interest in the programming language theory community. In particular, this is one of the central concerns of researchers working in mechanized meta-theory of programming languages.

4.2 A Central Challenge: Formalizing the Syntactic Meta-Theory

In studying Peter Andrews' formalism of \mathcal{Q}_0 , one notices that all logical derivations are made very explicit and carried out meticulously. In fact, one is immediately impressed by how such a high level of detail and accuracy is attained without the aid of the multitude of theorem provers and proof assistants available for our use today.

However, although logical derivations are made explicit, Andrews chooses to keep most syntactic derivations implicit in the meta-proofs of \mathcal{Q}_0 . This poses a challenge for our work on producing machine-checked proofs of the meta-theorems. Consider the following example of a syntactic derivation kept implicit in Andrews' formalism:

If \mathbf{A}' is the result of replacing all free occurrences of x by y in a well-formed formula \mathbf{A} , and if x does not occur free in \mathbf{A} , then we have that \mathbf{A}' is the same as \mathbf{A} .

Such syntactic lemmas are arguably intuitive and straightforward and therefore are understandably kept implicit even a formalism as thorough and meticulous as Andrews'. However, in a completely formal development such as ours all derivations, be they logical or syntactic, have to be made explicit to the theorem prover. Considering again the above example, we see that even the *statement* of such a straightforward lemma in a theorem prover entails the encoding and formalizing of syntactic notions such as variable occurrence, variable binding, and the equivalence of terms. \mathcal{Q}_0 's complex syntactic notions pose additional challenges. Typically, substitutions allowed by replacement rules of logical systems are non-linear, meaning all occurrences

are replaced, and variable capture avoiding. Substitution in \mathcal{Q}_0 , however, allows for variable capture in certain cases, and only replaces a single occurrence.

4.2.1 Previous work: The Sublogic in RSP

We briefly mention as an aside how the syntactic meta-theory of \mathcal{Q}_0 is encoded in our formalism in Rogue-Sigma-Pi (RSP) [13]. The approach we chose to take there was to encode an underlying logical system — what we called a *sublogic* — within our logical framework RSP. With the added expressiveness of this sublogic in our logical framework, we then formalize various syntactic relations among \mathcal{Q}_0 terms and variables, which allow us to build proofs of meta-theoretic properties resulting from these relations. The proofs of these syntactic lemmas are used to provide explicit syntactic derivations in the meta-proofs of \mathcal{Q}_0 's meta-theorems.

Our sublogic in RSP is a multi-sorted classical first-order natural deduction system. The primitive sorts of our sublogic are \mathcal{Q}_0 types, variables, constants, terms, as well as positions in terms and variable numbers. The sublogic permits quantification over all primitive sorts. Also, we take a small set of basic syntactic notions as primitive. Every derived syntactic notion is then defined by a first-order formula of our sublogic, and carrying out syntactic derivations corresponds to building proofs in our natural deduction system.

```

sl_o :: type.           # sublogic formula
sl_pf :: sl_o => type.  # sublogic proof

```

Therefore, a sublogic formula A is true if and only if the type `sl_pf A` is inhabited. Our development of the syntactic meta-theory of Coq is very similar to the approach mentioned here, except that we adopt the primitive intuitionistic logic of the Calculus of Constructions as our sublogic.

4.3 The Primitive Basis of \mathcal{Q}_0

4.3.1 \mathcal{Q}_0 types, constants and terms

In this section, we will discuss the embedding of the primitive basis of \mathcal{Q}_0 in the Calculus of Inductive Constructions. A presentation of \mathcal{Q}_0 types, variables, constants

and terms will run parallel with a discussion of their corresponding representations in the interactive theorem prover Coq.

The types of \mathcal{Q}_0 are defined recursively as shown below.

types t ::=	ι	<i>type of individuals</i>
	o	<i>type of truth values</i>
	$(\alpha\beta)$	<i>type of functions from elements of type β to elements of type α</i>

We bring to the reader's attention a peculiarity in the definition given above. One might find it counter-intuitive that the type of functions from elements of type β to elements of type α is denoted by $(\alpha\beta)$ instead of the seemingly more natural $(\beta\alpha)$. The motivation behind this choice of notation does become apparent when one considers the function application of a term of type $(\beta\alpha)$, denoted $A_{\alpha\beta}$, to an argument. First, note that our typing rules dictate that the argument to $A_{\alpha\beta}$ necessarily has type β and hence we denote it as B_β . Now, we see that the function application denoted by $A_{\alpha\beta}B_\beta$ in fact explicitly underscores the agreement of types, and we see how this is in some sense a better notation than $A_{\beta\alpha}B_\beta$.

We now give the definition of \mathcal{Q}_0 types as encoded in our formalism in Coq. We note the impressively close correspondence between the inductive mathematical definition presented in Andrews' formalism on paper, and the inductive definition entered into a theorem prover as shown below.

1. `Inductive q0_type : Set :=`
2. `I : q0_type`
3. `| 0 : q0_type`
4. `| Fun : q0_type -> q0_type -> q0_type.`

The Coq keywords `Inductive` and `Set` in Line 1 denote the start of an inductively defined set of objects. Lines 2 and 3 correspond to the base cases of the inductive definition; they encode 0-arity constructors `I` and `0` that construct objects representing the type of individuals and truth values respectively. Lastly, Line 4 defines the constructor for types of functions. The type of a function is indexed by two other types, the type of its domain and the type of its range; correspondingly, `Fun` takes in

two terms of type `q0_type` and returns a term of type `q0_type`.

With the exception of the abstraction operator λ and parentheses `[]`, the only primitive symbols of \mathcal{Q}_0 are variables and logical constants. For every type symbol α , we have a countably infinite set of variables of type α :

$$v_\alpha, v'_\alpha, v''_\alpha, v'''_\alpha, v''''_\alpha, \dots$$

Accordingly, \mathcal{Q}_0 variables in Coq are indexed not only by their types, but also by a natural number.

```
Inductive Var : q0_type -> Set :=
  V_ : forall t, nat -> Var t.
```

Therefore, `V_` is a constructor that takes in a type `t` and a natural number n and returns the n^{th} variable of type `t`. For example, `(V_ 0 1)` gives us the first variable of type `0`, and `(V_ (Fun 0 I) 5)` gives us the fifth variable of type `(Fun 0 I)`.

For every type α , we have a corresponding logical constant $\mathbb{Q}_{((o\alpha)\alpha)}$ denoting the identity relation between elements of type α . Also, there is the description operator (also known as a selection operator) $\iota_{(i(o\alpha))}$. First, we note that the semantics assigned to these logical constants do in fact correspond to the intuition given to us by their types. $\mathbb{Q}_{((o\alpha)\alpha)}$ is the curried version of a function that takes in two terms of type α and returns a truth value (we remind our readers once again that the types of functions should be read from right to left.) In other words, for terms A_α and B_α of type α , $(\mathbb{Q}_{((o\alpha)\alpha)}A_\alpha)B_\alpha$ is a term of type o , the type of truth values. The description operator $\iota_{(i(o\alpha))}$ takes in a function from individuals to truth values, and returns an individual. One may therefore view $\iota_{(i(o\alpha))}$ as a function mapping subsets of the set of all individuals to individuals — hence its name the *selection* operator. The encoding of $\mathbb{Q}_{((o\alpha)\alpha)}$ and $\iota_{(i(o\alpha))}$ in Coq is shown below.

```
Inductive Const : q0_type -> Set :=
  Q_ : forall t, Const (Fun (Fun 0 t) t)
  | Iota_ : Const (Fun I (Fun 0 I)).
```

Notice that the constant $\mathbb{Q}_{((o\alpha)\alpha)}$ is indexed by a type symbol α . Correspondingly, our constructor `Q_` is universally quantified over the set of all objects of type `q0_type`; for

example, the term $(Q_0 \text{ (Fun I 0)})$ is the identity relation between elements of type (ιo) and its own type is $(o(\iota o))(\iota o)$.

In \mathcal{Q}_0 , well-formed formulas, also known as *terms*, are indexed by their corresponding types. We shall adopt Andrews' convention of denoting well-formed formulas with uppercase letters subscripted by their types — for example, \mathbf{A}_α , \mathbf{B}_β and $\mathbf{C}_{\alpha\beta}$ represent well-formed formulas of types α , β and $(\alpha\beta)$ respectively. Further, we abbreviate the collection of all well-formed formulas of type α by wff_α . The type symbol o is usually elided, and we will use \mathbf{A} , \mathbf{B} , \mathbf{C} as syntactic variables for well-formed formulas of type o . The recursive definition of well-formed formulas is then given as follows.

1. A primitive variable or constant of type α is a wff_α .
2. $\mathbf{A}_{\alpha\beta}\mathbf{B}_\beta$ is a wff_α denoting the application of the function $\mathbf{A}_{\alpha\beta}$ to \mathbf{B}_β .
3. $\lambda x_\beta \mathbf{A}_\alpha$ is a $wff_{(\alpha\beta)}$ denoting a lambda abstraction.

Again, we note that our encoding of the logical constants of \mathcal{Q}_0 in Coq is virtually isomorphic to the mathematical definition given above.

```

Inductive Trm : q0_type -> Set :=
  | _v : forall t, Var t -> Trm t
  | _c : forall t, Const t -> Trm t
  | Lambda : forall s t, Var s -> Trm t -> Trm (Fun t s)
  | Apply : forall s t, Trm (Fun s t) -> Trm t -> Trm s.

```

The constructors `_v` and `_c` allow us to cast variables and constants as terms. For example, the following are the terms (with type `Trm`) corresponding to the two logical constants of \mathcal{Q}_0 .

```

Definition Iota := _c Iota_.
Definition Q := fun (t : q0_type) => _c (Q_ t).

```

Similarly, `_v (V_ 0 3)` and `_v (V_ (Fun 0 I) 15)` have type `Trm` in our formalism.

```

Check (_v (V_ 0 3)).
_v (V_ 0 3) : Trm 0
Check (_v (V_ (Fun 0 I) 15)).
_v (V_ (Fun 0 I) 15) : Trm (Fun 0 I).

```

4.3.2 Definitions of Logical Operators in \mathcal{Q}_0

As mentioned in the previous section, the only primitive non-trivial symbols of \mathcal{Q}_0 are variables and the logical constants $\mathbf{Q}_{((o\alpha)\alpha)}$ and $\iota_{(\iota(o\iota))}$. All other logical operators, such as \wedge , \vee , \supset , and \sim are defined in terms of primitive symbols. We list here all abbreviations and definitions of logical operators in \mathcal{Q}_0 . The square dot \cdot represents a left bracket, where the corresponding right bracket is as far right as possible without violating any other pairings already present.

$[\mathbf{A}_\alpha = \mathbf{B}_\alpha]$	stands for	$[\mathbf{Q}_{o\alpha\alpha} \mathbf{A}_\alpha \mathbf{B}_\alpha]$
$[\mathbf{A} \equiv \mathbf{B}]$	stands for	$[\mathbf{Q}_{ooo} \mathbf{A} \mathbf{B}]$
T_o	stands for	$[\mathbf{Q}_{ooo} = \mathbf{Q}_{ooo}]$
F_o	stands for	$[\lambda x_o T] = [\lambda x_o x_o]$
$\Pi_{o(\alpha\alpha)}$	stands for	$[\mathbf{Q}_{o(o\alpha)(o\alpha)} [\lambda x_\alpha T]]$
$[\forall \mathbf{x}_\alpha \mathbf{A}]$	stands for	$[\Pi_{o(o\alpha)} [\lambda \mathbf{x}_\alpha \mathbf{A}]]$
\wedge_{ooo}	stands for	$[\lambda x_o \lambda y_o \cdot [\lambda g_{ooo} \cdot g_{ooo} T T] = [\lambda g_{ooo} \cdot g_{ooo} x_o y_o]]$
$[\mathbf{A} \wedge \mathbf{B}]$	stands for	$[\wedge_{ooo} \mathbf{A} \mathbf{B}]$
\supset_{ooo}	stands for	$[\lambda x_o \lambda y_o \cdot x_o = \cdot x_o \wedge y_o]$
$[\mathbf{A} \supset \mathbf{B}]$	stands for	$[\supset_{ooo} \mathbf{A} \mathbf{B}]$
\sim_{oo}	stands for	$[\mathbf{Q}_{ooo} F]$
\vee_{ooo}	stands for	$[\lambda x_o \lambda y_o \cdot \sim \cdot [\sim x_o] \wedge [\sim y_o]]$
$[\mathbf{A} \vee \mathbf{B}]$	stands for	$[\vee_{ooo} \mathbf{A} \mathbf{B}]$
$[\exists \mathbf{x}_\alpha \mathbf{A}]$	stands for	$[\sim \forall \mathbf{x}_\alpha \sim \mathbf{A}]$
$[\mathbf{A}_\alpha \neq \mathbf{B}_\alpha]$	stands for	$[\sim \cdot \mathbf{A}_\alpha = \mathbf{A}_\alpha]$

The above abbreviations and definitions have all be formalized in Coq. Due to space considerations, we will only give the first few encodings here.

```

Definition Equals (t : q0_type)
  := fun (A:Trm t)(B:Trm t) => Apply (Apply (Q t) A) B.
Definition Equiv
  := fun (A B:Trm 0) => Apply (Apply (Q 0) A) B.
Definition True := Equals (Q 0) (Q 0).
Definition False
  := Equals (Lambda (V 0) True) (Lambda (V 0) (_v (V 0))).

```

Definition Pi (t : q0_type)

:= Apply (Q (Fun 0 t)) (Lambda (V t) True).

Definition Forall (t : q0_type)

:= fun (x : Var t)(A:Trm 0) => Apply (Pi t) (Lambda x A).

4.3.3 Axioms and Inference Rules

\mathcal{Q}_0 has five axioms, three of which (2^α , $3^{\alpha\beta}$ and $4_1 - 4_5$) are axiom schemata.

Axioms for \mathcal{Q}_0

- (1) $g_{oo}T_o \wedge g_{oo}F_o = \forall x_o \cdot g_{oo}x_o$
- (2^α) $[x_\alpha = y_\alpha] \supset \cdot h_{o\alpha}x_\alpha = h_{o\alpha}y_\alpha$
- ($3^{\alpha\beta}$) $f_{\alpha\beta} = g_{\alpha\beta} = \forall x_\beta \cdot f_{\alpha\beta}x_\beta = g_{\alpha\beta}x_\beta$
- (4₁) $[\lambda x_\alpha \mathbf{B}_\beta] \mathbf{A}_\alpha = \mathbf{B}_\beta$ where \mathbf{B}_β is a primitive constant or variable distinct from x_α
- (4₂) $[\lambda x_\alpha x_\alpha] \mathbf{A}_\alpha = \mathbf{A}_\alpha$
- (4₃) $[\lambda x_\alpha \cdot \mathbf{B}_{\beta\gamma} \mathbf{C}_\gamma] \mathbf{A}_\alpha = [[\lambda x_\alpha \mathbf{B}_{\beta\gamma}] \mathbf{A}_\alpha][[\lambda x_\alpha \mathbf{C}_\gamma] \mathbf{A}_\alpha]$
- (4₄) $[\lambda x_\alpha \cdot \lambda y_\gamma \mathbf{B}_\delta] \mathbf{A}_\alpha = [\lambda y_\gamma \cdot [\lambda x_\alpha \mathbf{B}_\delta] \mathbf{A}_\alpha]$ where y_γ is distinct from x_α and from all variables in \mathbf{A}_α
- (4₅) $[\lambda x_\alpha \cdot \lambda x_\alpha \mathbf{B}_\delta] \mathbf{A}_\alpha = [\lambda x_\alpha \mathbf{B}_\delta]$
- (5) $\iota_{\iota(o\iota)}[\mathbf{Q}_{o\iota}y_\iota] = y_\iota$

Axiom 1 states that we are working in a bimodal logic. That is, *Truth* and *Falsehood* are the only truth values in our logical system, or equivalently, T_o and F_o are the only well-formed formulas of type o . Axiom Schemata 2 and 4₁–4₅ give us basic properties about equality and λ , respectively. Axiom Schema $3^{\alpha\beta}$ states the \mathcal{Q}_0 equivalent of the Axiom of Extensionality, while Axiom 5 gives us the \mathcal{Q}_0 equivalent of the Axiom of Descriptions.

After instantiating the variables g_{oo} , x_o and meta-variables x_α , y_α , $h_{o\alpha}$, $f_{\alpha\beta}$, $g_{\alpha\beta}$, Axioms 1, 2^α and $3^{\alpha\beta}$ are dealt with routinely.

Axiom axiom_1 :

```

Pf (Equals (And (Apply (_v g_oo) True) (Apply (_v g_oo) False))
  (Forall 0 x_o (Apply (_v g_oo) (_v x_o)))).
Axiom axiom_2a : forall (a : q0_type),
Pf (Implies (Equals (_v (x_a a)) (_v (y_a a)))
  (Equals (Apply (_v (h_oa a)) (_v (x_a a)))
    (Apply (_v (h_oa a)) (_v (y_a a))))).
Axiom axiom_3ab : forall (a b : q0_type),
Pf (Equals (Equals (_v (f_ab a b)) (_v (g_ab a b)))
  (Forall _ (x_a b)
    (Equals (Apply (_v (f_ab a b)) (_v (x_a b)))
      (Apply (_v (g_ab a b)) (_v (x_a b)))))).

```

Axiom Schema 4₁, however, is not as straightforward. In order to state the restriction that \mathbf{B}_β is a primitive constant or *variable distinct from* \mathbf{x}_α , we need a notion of equality between variables. Fortunately, this is simple enough. Variables in \mathcal{Q}_0 are indexed by a type and a natural number; hence, two variables are considered equivalent if and only if they are of the same type and are indexed by the same number. Now equality of variables is reduced to equality of types and natural numbers.

```

Inductive eqvar : forall a b, Var a -> Var b -> Prop :=
  eqvar_ax : forall a b n1 n2,
    a = b ->
    n1 = n2 ->
    eqvar a b (V_ a n1) (V_ b n2)).

```

Two variables are considered different simply when they are not equal.

```

Definition diffvar (a b: q0_type)(A : Var a)(B : Var b)
:= ~(eqvar A B).

```

With this, we may now state Axiom Schema 4₁. We split 4₁ up into 4_{1v} and 4_{1c}, mirroring the case split on whether \mathbf{B}_α is a variable or a constant.

```

Axiom axiom_41v :
  forall (a b : q0_type), forall (x : Var a),
  forall (A : Trm a), forall (v: Var b),
  diffvar x v ->
  Pf (Equals (Apply (Lambda x (_v v)) A) (_v v))).

```

Axiom axiom_41c :
 forall (a b : q0_type), forall (x : Var a),
 forall (A : Trm a), forall (c : Const b),
 Pf (Equals (Apply (Lambda x (_c c)) A) (_c c)).

The statement of 4₄ poses yet another challenge.

(4₄) $[\lambda x_\alpha . \lambda y_\gamma \mathbf{B}_\delta] \mathbf{A}_\alpha = [\lambda y_\gamma . [\lambda x_\alpha \mathbf{B}_\delta] \mathbf{A}_\alpha]$ where y_γ is distinct from x_α
 and from all variables in \mathbf{A}_α

Equivalent to formalizing the notion that y_γ is distinct from all variables in \mathbf{A}_α is being able to state the following property of \mathbf{A}_α

forall t, forall v : Var t, occurs (_v v) A => diffvar v y

We will set this aside and press ahead for now, noting that this is the first indication that an encoding of the syntactic meta-theory of \mathcal{Q}_0 is necessary before we can proceed. By presenting more examples of the same flavor, we hope to make the case that these syntactical issues can no longer be dealt with in an online fashion, but instead, we need to develop and formalize a complete syntactic meta-theory.

\mathcal{Q}_0 's single rule of inference is a rule of replacement which allows for variable capture. We remind our readers that $\mathbf{A}, \mathbf{B}, \mathbf{C}$ etc. are abbreviations for the *wff*s $\mathbf{A}_o, \mathbf{B}_o, \mathbf{C}_o$ etc.

Rule R From \mathbf{C} and $\mathbf{A}_\alpha = \mathbf{B}_\alpha$ to infer the result of replacing one occurrence of \mathbf{A}_α in \mathbf{C} by an occurrence of \mathbf{B}_α , provided that the occurrence of \mathbf{A}_α in \mathbf{C} is not immediately preceded by λ .

We immediately see that the statement of Rule R poses several challenges. There are three non-trivial syntactic notions, all of which have to be first explicitly dealt with in our formalism before we can even state the rule.

1. \mathbf{A}_α is a subterm of \mathbf{C} at position p .
2. Replacing \mathbf{A}_α by \mathbf{B}_α in \mathbf{C} at position p gives us \mathbf{D} .
3. The occurrence of \mathbf{A}_α in \mathbf{C} is not immediately preceded by λ . Equivalently, if \mathbf{A}_α is a subterm of \mathbf{C} at position p , then p is not a binding position in \mathbf{C} .

We note that the statement of Rule R is in fact trivial after we have formalized these three syntactic judgments – this should not come as a surprise as the notion of linear replacement as captured by Rule R is a purely syntactic one.

```
Axiom Rule_R :
  forall a : q0_type, forall A B : Trm a,
  forall p : Pos, forall C D : Trm 0,
  Pf C ->
  Pf (Equals A B) ->
  replace A B p C D ->
  ~ pos_binding p C ->
  Pf D.
```

For a more elaborate example, consider the following generalization of Rule R.

Rule R' From $\mathcal{H} \vdash \mathbf{A}_\alpha = \mathbf{B}_\alpha$ and $\mathcal{H} \vdash \mathbf{C}$ to infer the result of replacing one occurrence of \mathbf{A}_α in \mathbf{C} by an occurrence of \mathbf{B}_α , provided the occurrence of \mathbf{A}_α in \mathbf{C} is not immediately preceded by λ and the occurrence of \mathbf{A}_α in \mathbf{C} is not in a wf part $[\lambda x_\beta E_\gamma]$ of \mathbf{C} , where x_β is free in a member of \mathcal{H} and free in $[\mathbf{A}_\alpha = \mathbf{B}_\alpha]$

We hope it is now clear that in order for our development of \mathcal{Q}_0 in Coq to proceed, a formalization of \mathcal{Q}_0 's syntactic meta-theory is necessitated. In fact, since the syntactic details of \mathcal{Q}_0 were implicitly assumed in Peter Andrews' presentation, we have to first develop the syntactic meta-theory on paper, before formalizing it in Coq. In this way, we will be able to provide formal proofs of syntactic judgments (including but not limited to those mentioned above), which are then combined with logical derivations to provide *completely* formal proofs of \mathcal{Q}_0 's meta-theorems.

4.4 The Syntactic Meta-Theory

In our syntactic meta-theory, we take several elementary notions as primitive: type-equivalence, variable-equivalence, constant-equivalence, term-equivalence, and position-equivalence, the subterm relationship and the prefix relationship between two positions. All other syntactic notions are derived from these basic primitive notions. Our approach is built on the following ideal: All primitive notions should be simple

enough that their correctness can be checked easily. Any notions entailing non-trivial definitions (variable binding, for example) are derived from primitive notions.

We will walk through an example of defining and formalizing a primitive syntactic notion. The subterm relationship is defined by a ternary judgment – \mathbf{A} is a subterm of \mathbf{B} at position p , which we will denote as $A \prec_p B$. First, we state (informally, on paper) the axioms and inference rules corresponding to the subterm judgment. We will let ϵ denote the empty position and $0 \cdot p$ and $1 \cdot p$ denote the position we get by prefixing p by 0 and 1 (left and right) respectively.

$$\frac{}{A \prec_{\epsilon} A} \text{subtrm-empty}$$

$$\frac{T \prec_p M}{T \prec_{0 \cdot p} M N} \text{subtrm-apply0} \qquad \frac{T \prec_p N}{T \prec_{1 \cdot p} M N} \text{subtrm-apply1}$$

$$\frac{T \prec_p x}{T \prec_{0 \cdot p} \lambda x.M} \text{subtrm-lambda0} \qquad \frac{T \prec_p N}{T \prec_{1 \cdot p} \lambda x.M} \text{subtrm-lambda1}$$

Each rule then corresponds to a case in our inductive definition of `subtrm` – axioms correspond to base cases, and inference rules correspond to inductive constructors.

```

Inductive subtrm : forall a b, Trm a -> Trm b -> Pos -> Prop :=
  subtrm_empty : forall a A, subtrm a a A A empty
| subtrm_apply0 : forall a b t M N T p,
  subtrm t (Fun a b) T M p ->
  subtrm _ _ T (Apply M N) (P0 p)
| subtrm_apply1 : forall a b t M N T p,
  subtrm t b T N p ->
  subtrm _ a T (Apply M N) (P1 p)
| subtrm_lambda0 : forall a b t x M T p,
  subtrm t a T (_v x) p ->
  subtrm t (Fun b a) T (Lambda x M) (P0 p)
| subtrm_lambda1 : forall a b t x M T p,
  subtrm t b T M p ->

```

subtrm t (Fun b a) T (Lambda x M) (P1 p).

Naturally, building proofs of such syntactic judgments can be tedious and burdensome. For instance, consider the following.

$$Q_{ooo} \prec_{011111001} [\lambda x_o \lambda y_o \cdot [\lambda g_{ooo} \cdot g_{ooo}[Q_{ooo} = Q_{ooo}][Q_{ooo} = Q_{ooo}]] = [\lambda g_{ooo} \cdot g_{ooo} x_o y_o]$$

A proof of the above judgment, if it exists, should be easy to find mechanically. In fact, Coq has language support for meta-level *tactics* that automate the building of such proofs of judgments. We are able to guarantee total correctness of our tactics. That is, the tactics we write all have the following soundness and completeness guarantees.

Soundness Every proof a tactic builds is valid.

Completeness If a proof exists, the tactics are guaranteed to find it.

We write a corresponding tactic for each judgment in our syntactic meta-theory. For example, the tactic for subterm takes in arbitrary types α and β , terms \mathbf{A}_α and \mathbf{B}_β , a position p , and builds a proof of $\mathbf{A}_\alpha \prec_p \mathbf{B}_\beta$ if one exists. As our syntactic notions get more complex and intricate, we have *tacticals* that call tactics as subroutines. Tactics and tacticals therefore help relieve the burden of having to build lengthy proofs by hand.

Having encoded a full syntactic meta-theory of \mathcal{Q}_0 in our formalism in Coq, we are now able to state many syntactic notions integral to the development of the theory of \mathcal{Q}_0 . For example, let us consider again Rule R', the generalization of Rule R.

Rule R' From $\mathcal{H} \vdash \mathbf{A}_\alpha = \mathbf{B}_\alpha$ and $\mathcal{H} \vdash \mathbf{C}$ to infer the result of replacing one occurrence of \mathbf{A}_α in \mathbf{C} by an occurrence of \mathbf{B}_α , provided the occurrence of \mathbf{A}_α in \mathbf{C} is not immediately preceded by λ and the occurrence of \mathbf{A}_α in \mathbf{C} is not in a wf part $[\lambda x_\beta E_\gamma]$ of \mathbf{C} , where x_β is free in a member of \mathcal{H} and free in $[\mathbf{A}_\alpha = \mathbf{B}_\alpha]$

Rule R' can now be formally stated in Coq as follows.

1. Axiom Rule_R' :
2. forall H, forall a, forall A B : Trm a, forall C D p,
3. HypPf H (Equals A B) ->
4. HypPf H C ->

5. `replace A B p C D ->`
6. `~ pos_binding p C ->`
7. `forall p' : Pos, forall x, forall E,`
8. `(subterm (Lambda x E) C p') /\ (occurs A E) ->`
9. `(forall h, member h H -> ~ free_in x h)`
10. `\ / ~ (free_in x (Equals A B))) ->`
11. `HypPf H D.`

Lines 5-10 correspond to syntactic properties of our input. We have a tactical for the Rule R' that calls tactics corresponding are called for each of these syntactic properties. If proofs are successfully found and built for all of them, we are then allowed to infer as a result of Rule R' the linear replacement of one occurrence of A_α in $\mathcal{H} \vdash C$ by B_α .

We briefly mention here an interesting challenge that arises in the formalization of \mathcal{Q}_0 's syntactic meta-theory. In the process of proving various syntactic properties of \mathcal{Q}_0 , we have on several occasions needed the following seemingly intuitive fact about dependent types: If two dependent types are equal, then their corresponding indexing terms have to be equal too. In the language of Coq, such an injectivity property can be stated as follows.

```

Lemma projS2_eq :
  forall (A : Set)(P : A -> Set)(x : A) (p1 p2 : P x),
    existS P x p1 = existS P x p2 -> p1 = p2.

```

where `existS` is a construct for building nested subset types

```

Inductive sigS (A : Set) (P : A -> Set) : Set :=
  existS : forall x : A, P x -> sigS A P.
Implicit Arguments sigS [A].

```

Somewhat surprisingly, Lemma `projS2_eq` above is in fact independent of Coq's type theory, the reason being that it is equivalent to Streicher's K axiom. Hence, we have had to include this injectivity property as an axiom in our formalism.

4.5 The Meta-Theorems of \mathcal{Q}_0

Having developed and formalized the syntactic meta-theory of \mathcal{Q}_0 , we have since proceeded to encoding formal proofs of \mathcal{Q}_0 's meta-theorems in Coq. Examples of

meta-theorems proved in our formalism include reflexivity of equality, basic equality rules, general and restricted versions of substitution in \mathcal{Q}_0 (such as β -reduction) as well as η -conversion. Many of these meta-theorems rely heavily on the inductive reasoning power of Coq and the Calculus of Inductive Constructions. For example, consider the following statement of a form of β -reduction in \mathcal{Q}_0 . Theorem 5203 states that

$$\vdash [\lambda \mathbf{x}_\alpha \mathbf{B}_\beta] \mathbf{A}_\alpha = \mathbf{S}_{\mathbf{A}_\alpha}^{\mathbf{x}_\alpha} \mathbf{B}_\beta \text{ provided no variable in } \mathbf{A}_\alpha \text{ is bound in } \mathbf{B}_\beta$$

where $\mathbf{S}_{\mathbf{A}_\alpha}^{\mathbf{x}_\alpha} \mathbf{B}_\beta$ denotes the result of substituting free occurrences of \mathbf{x}_α by \mathbf{A}_α in \mathbf{B}_β . The proof of this theorem is done by induction on the form of \mathbf{B}_β , or equivalently, the number of occurrences of $[$ in \mathbf{B}_β .

Having completed this phase of the project, proving the meta-theorems of \mathcal{Q}_0 has become the main emphasis of our work. However, as new syntactic notions arise in the statement and proofs of \mathcal{Q}_0 's meta-theorems, we certainly expect that the syntactic meta-theory we have developed will be revisited frequently and perhaps supplemented from time to time. It is our hope that our work will culminate in the proving of the Deduction Theorem of \mathcal{Q}_0 .

Chapter 5

Conclusion

Informal proofs mathematicians write are susceptible to logical errors – routine logical steps are left out, and a great amount of context is assumed on the part of the reader. Traditionally, the correctness of a proof has been verified by peer review, where the reviewer fills in some of the logical gaps herself and skips over the others. This process cannot provide us with absolute guarantees of the total correctness proofs. In light of this, there has been significant interest in producing *formal proofs* of mathematical theorems. Formal proofs are proofs in which every intermediate logical step is supplied, and no appeal is made to intuition. Naturally, such proofs can become over symbolic and syntactic, and hence burdensome to build by hand. Fortunately, the advent of automated theorem provers and interactive proof assistants have made the building of formal proofs a tractable endeavor. In fact, advances in such technology have gone a long way in making the process akin to writing an informal proof on paper. Drawing on ideas from Type Theory, Computational Logic and the theory of Automated Deduction, these theorem provers and proof assistants are able to guarantee the total correctness of our proofs.

In this thesis, we have discussed our contributions to the formalization of mathematics. We discussed our effort to formalize the meta-theory of Peter Andrews' classical higher-order logic \mathcal{Q}_0 based on typed lambda calculus. There are several reasons why a formal proof of \mathcal{Q}_0 's correctness is of significant interest and importance. First, \mathcal{Q}_0 is the formal framework within which Peter Andrews develops core areas in the foundations of mathematics, including cardinals and the axiom of infinity, Peano's postulates and primitive recursive functions. Second, \mathcal{Q}_0 is an elegant logical system with significant aesthetic appeal – it is a minimal logic based only on equality, with

all its meta-theory built on just 5 axioms and 1 inference rule. Third, the complex syntax of \mathcal{Q}_0 makes it impossible for us to check its meta-theorems by hand. Such challenges lend themselves very naturally to the use of interactive proof assistants and automated theorem provers. Last, issues such as variable binding and replacement which we have to deal with explicitly in our formalism are of contemporary interest in the programming language community. In particular, this has become a primary concern of researchers working in mechanized meta-theory of programming languages.

Our formalization is carried out in the interactive proof assistant Coq, based on the Calculus of Inductive Construction. The Calculus of Inductive Construction is an extension of Coquand and Huet’s Calculus of Constructions with support for inductive data types. Both systems are based on higher-order typed lambda calculus, and are designed for the building of proofs in full intuitionistic predicate calculus. Coq has been a major force in the formalization of mathematics; the full formal proofs of the Four Color Theorem and Gödel’s Incompleteness Theorem, among hundreds of others, have both been encoded in Coq.

A central challenge that arose in the formalizing of \mathcal{Q}_0 was having to explicitly encode and reason about its various syntactic notions. Although Peter Andrews was impressively careful in his development of \mathcal{Q}_0 ’s logical meta-theory, providing all logical derivations in great detail, the *syntactic* meta-theory was kept implicit in his presentation. In a completely formal development such as ours, all derivations, logical and syntactic, have to be made explicit; even just the statement of many of \mathcal{Q}_0 ’s meta-theorems necessitate reasoning about complex syntactic notions. Therefore, we first developed and encoded the syntactic meta-theory in our formalism in Coq. Within this framework, we were then able to encode and reason about \mathcal{Q}_0 ’s syntactic notions such as scope, variable binding, and replacement. This allowed us to explicitly carry out syntactic derivations kept implicit in Andrews’ presentation and to provide *completely* formal proofs of \mathcal{Q}_0 ’s meta-theorems. As far as we know, this thesis presents the first effort to formalize Peter Andrews’ logical system \mathcal{Q}_0 .

Appendix

Attached in this appendix is the Coq source code for our formalization of \mathcal{Q}_0 . As this is still very much work in progress, the most updated version may be obtained either online at <http://cl.cse.wustl.edu>, or by contacting the author at lytan@wustl.edu.

The Core of \mathcal{Q}_0 : Types, Variables, Constants and Terms

```
(* Q0 types *)
Inductive q0_type : Set :=
  I : q0_type
| 0 : q0_type
| Fun : q0_type -> q0_type -> q0_type.

(* Q0 variables *)
Inductive Var : q0_type -> Set :=
  V_ : forall t, nat -> Var t.

(* Q0 constants *)
Inductive Const : q0_type -> Set :=
  Iota_ : Const (Fun I (Fun 0 I))
| Q_ : forall t, Const (Fun (Fun 0 t) t).

(* Q0 terms *)
Inductive Trm : q0_type -> Set :=
  _v : forall t, Var t -> Trm t
| _c : forall t, Const t -> Trm t
| Lambda : forall s t, Var s -> Trm t -> Trm (Fun t s)
| Apply : forall s t, Trm (Fun s t) -> Trm t -> Trm s.

Implicit Arguments _v.
Implicit Arguments _c.
Implicit Arguments Lambda.
Implicit Arguments Apply.

Definition V := fun (t : q0_type) => V_ t 0.
```

Definition Iota := _c Iota_.

Definition Q := fun (t : q0_type) => _c (Q_ t).

Basic Syntactic Notions

(* Equality between types, variables, constants and terms *)

Inductive eqtp : q0_type -> q0_type -> Prop :=

 eqtp_ax : forall a b : q0_type,
 a = b ->
 eqtp a b.

Definition difftp (a b : q0_type) := ~(eqtp a b).

Inductive eqvar : forall a b, Var a -> Var b -> Prop :=

 eqvar_ax : forall a b n1 n2,
 eqtp a b -> n1 = n2 ->
 eqvar a b (V_ a n1) (V_ b n2).

Definition diffvar (a b : q0_type)(A : Var a)(B : Var b)
:= ~(eqvar A B).

Inductive eqtrm : forall a b, Trm a -> Trm b -> Prop :=

 eqtrm_var : forall a b v1 v2,
 eqvar v1 v2 ->
 eqtrm a b (_v v1) (_v v2)
| eqtrm_const: forall a b, forall A : Const a, forall B : Const b,
 eqtp a b ->
 eqtrm a b (_c A) (_c B)
| eqtrm_apply: forall a b c d M1 N1 M2 N2,
 eqtrm (Fun a b) (Fun c d) M1 M2 ->
 eqtrm _ _ N1 N2 ->
 eqtrm _ _ (Apply M1 N1) (Apply M2 N2)
| eqtrm_lambda :
 forall a b c d,
 forall x : Var a, forall y : Var c, forall M1 M2,
 eqvar x y ->
 eqtrm b d M1 M2 ->
 eqtrm _ _ (Lambda x M1) (Lambda y M2).

Definition difftrm (a b : q0_type) (A : Trm a) (B : Trm b) :=
 ~(eqtrm A B).

```

Implicit Arguments eqvar.
Implicit Arguments diffvar.
Implicit Arguments eqtrm.
Implicit Arguments difftrm.

```

```

Theorem eqtrm_imp_eqtp :
  forall a b, forall A : Trm a, forall B : Trm b,
    eqtrm A B -> eqtp a b.

```

Proof.

```

  intros.
  induction H.
  elim H.
  auto.
  assumption.
  inversion IHeqtrm1.
  assumption.
  elim H.
  intros.
  exact (eqtp_Fun _ _ _ _ IHeqtrm H1).

```

Qed.

(* Positions *)

```

Inductive Pos : Set :=
  empty : Pos
| P0 : Pos -> Pos
| P1 : Pos -> Pos.

```

```

Inductive eqpos : Pos -> Pos -> Prop :=
  eqpos_empty : eqpos empty empty
| eqpos_0 : forall p1 p2, eqpos p1 p2 -> eqpos (P0 p1) (P0 p2)
| eqpos_1 : forall p1 p2, eqpos p1 p2 -> eqpos (P1 p1) (P1 p2).
Definition diffpos (p1 p2 : Pos) := ~ eqpos p1 p2.

```

```

Inductive pos_left : Pos -> Pos -> Prop :=

```

```

pos_left_empty : pos_left empty (P0 empty)
| pos_left_0 : forall p1 p2,
  pos_left p1 p2 -> pos_left (P0 p1) (P0 p2)
| pos_left_1 : forall p1 p2,
  pos_left p1 p2 -> pos_left (P1 p1) (P1 p2).

```

```

Inductive prefix : Pos -> Pos -> Prop :=
  prefix_empty : forall p, prefix empty p
| prefix_0 : forall p1 p2, prefix p1 p2 -> prefix (P0 p1) (P0 p2)
| prefix_1 : forall p1 p2, prefix p1 p2 -> prefix (P1 p1) (P1 p2).

```

```

Definition disjoint (p1 p2 : Pos)
  := ~ (prefix p1 p2) /\ ~ (prefix p2 p1).

```

(* Subterm *)

```

Inductive subterm : forall a b, Trm a -> Trm b -> Pos -> Prop :=
  subterm_empty : forall a A, subterm a a A A empty
| subterm_apply0 : forall a b t M N T p,
  subterm t (Fun a b) T M p ->
  subterm _ _ T (Apply M N) (P0 p)
| subterm_apply1 : forall a b t M N T p,
  subterm t b T N p ->
  subterm _ a T (Apply M N) (P1 p)
| subterm_lambda0 : forall a b t x M T p,
  subterm t a T (_v x) p ->
  subterm t (Fun b a) T (Lambda x M) (P0 p)
| subterm_lambda1 : forall a b t x M T p,
  subterm t b T M p ->
  subterm t (Fun b a) T (Lambda x M) (P1 p).

```

Implicit Arguments subterm.

```

Definition pos_binding (p : Pos)(a : q0_type)(A : Trm a) :=
  exists p2, exists s, exists t,
  exists x : Var s, exists M : Trm t,
  (pos_left p2 p) /\ (subterm (Lambda x M) A p2).

```

Implicit Arguments pos_binding.

Definition occurs (a t : q0_type)(x : Var a)(T : Trm t) :=
exists p, subtrm (_v x) T p.

Implicit Arguments occurs.

Definition replace

(a t : q0_type)(A B : Trm a)(p : Pos)(T1 T2 : Trm t) :=
(subtrm A T1 p) /\ (subtrm B T2 p) /\ forall p', (disjoint p p' ->
forall q, forall Q : Trm q,
(subtrm Q T1 p' <-> subtrm Q T2 p')).

Implicit Arguments replace.

Definition bound (a b : q0_type)(x : Var a)(B : Trm b) :=
exists p, exists t, exists M : Trm t, (subtrm (Lambda x M) B p).

Implicit Arguments bound.

Definition free (a b : q0_type)(x : Var a)(B : Trm b) :=
occurs x B /\ ~ bound a b x B.

Implicit Arguments free.

Definition var_in (t a : q0_type) (x : Var t) (A : Trm a) :=
exists p, subtrm (_v x) A p.

Implicit Arguments var_in.

Definition var_in_ (t a : q0_type) (x : Var t) (A : Trm a) (p : Pos) :=
subtrm (_v x) A p.

Implicit Arguments var_in_.

Definition free_in (t a : q0_type) (x : Var t) (A : Trm a) :=
exists p, (var_in_ x A p /\
forall p', forall t2 : q0_type, forall M : Trm t2,
prefix p' p -> ~ (subtrm (Lambda x M) A p')).

Implicit Arguments free_in.

Definition free_in_ (t a: q0_type) (x: Var t) (A: Trm a) (p: Pos) :=
 (var_in_ x A p /\
 forall p', forall t2: q0_type, forall M: Trm t2,
 prefix p' p -> ~ (subtrm (Lambda x M) A p')).
 Implicit Arguments free_in_.

Definition bound_in (t a: q0_type) (x: Var t) (A: Trm a) :=
 exists p, (var_in_ x A p /\
 exists p', exists t2:q0_type, exists M: Trm t2,
 prefix p' p /\ (subtrm (Lambda x M) A p')).
 Implicit Arguments bound_in.

Definition bound_in_ (t a: q0_type) (x: Var t) (A: Trm a) (p: Pos):=
 (var_in_ x A p /\
 exists p', exists t2:q0_type, exists M: Trm t2,
 prefix p' p /\ (subtrm (Lambda x M) A p')).
 Implicit Arguments bound_in_.

Inductive replace_all :
 forall a b, Var a -> Trm a -> Trm b -> Trm b -> Prop :=
 ra_eqvar : forall a x A,
 replace_all a a x A (_v x) A
| ra_diffvar : forall a b x A y,
 diffvar x y ->
 replace_all a b x A (_v y) (_v y)
| ra_const : forall a b x A c,
 replace_all a b x A (_c c) (_c c)
| ra_lambda_eqvar : forall a b x A M,
 replace_all a (Fun b a) x A (Lambda x M) (Lambda x M)
| ra_lambda_diffvar : forall a b c x A y M M',
 diffvar x y ->
 replace_all a b x A M M' ->
 replace_all a (Fun b c) x A (Lambda y M) (Lambda y M')
| ra_apply : forall a b c x A M N M' N',
 replace_all a (Fun b c) x A M M' ->

```
replace_all a c x A N N' ->  
replace_all a b x A (Apply M N) (Apply M' N').  
Implicit Arguments replace_all.
```

Logical Operators and Abbreviations

Definition Equals (t : q0_type)

```
:= fun (A:Trm t)(B:Trm t) => Apply (Apply (Q t) A) B.
```

Implicit Arguments Equals.

Definition Equiv

```
:= fun (A B:Trm O) => Apply (Apply (Q O) A) B.
```

Definition True := Equals (Q O) (Q O).

Definition False

```
:= Equals (Lambda (V O) True) (Lambda (V O) (_v (V O))).
```

Definition Pi (t : q0_type)

```
:= Apply (Q (Fun O t)) (Lambda (V t) True).
```

Definition Forall (t : q0_type)

```
:= fun (x : Var t)(A:Trm O) => Apply (Pi t) (Lambda x A).
```

Definition g_ooo := V (Fun (Fun O O) O).

Definition x_o := V O.

Definition y_o := V_ O 1.

Definition And_

```
:= Lambda x_o (Lambda y_o
  (Equals (Lambda g_ooo
    (Apply (Apply (_v g_ooo) True) True))
    (Lambda g_ooo
      (Apply (Apply (_v g_ooo) (_v x_o)) (_v y_o)))))).
```

Definition And := (fun (A B:Trm O) => Apply (Apply And_ A) B).

Definition Implies_

```
:= Lambda x_o (Lambda y_o
  (Equals (_v x_o) (And (_v x_o) (_v y_o)))).
```

Definition Implies

```
:= (fun (A B:Trm O) => Apply (Apply Implies_ A) B).
```

(* Abbreviations contd: De Morgan's *)

Definition Not_ := Apply (Q O) False.

Definition Not := fun (A:Trm O) => Apply Not_ A.

Definition Or_

```
:= Lambda x_o (Lambda y_o
  (Not (And (Not (_v x_o)) (Not (_v y_o))))).
```

Definition Or := fun (A B:Trm 0) => Apply (Apply Or_ A) B.

Definition Exists (t : q0_type)

:= fun (x : Var t)(A : Trm 0) => Not (Forall t x (Not A)).

Definition Neq (t : q0_type)

:= fun (A B : Trm 0) => Not (Equals A B).

The Axioms of \mathcal{Q}_0

Variable Pf : Trm 0 -> Set.

Definition g_oo := V (Fun 0 0).

Definition x_a (a : q0_type) := V a.

Definition y_a (a : q0_type) := V_ a 1.

Definition h_oa (a : q0_type) := V (Fun 0 a).

(* Law of Bivalence *)

Axiom axiom_1 :

Pf (Equals (And (Apply (_v g_oo) True) (Apply (_v g_oo) False))
(Forall 0 x_o (Apply (_v g_oo) (_v x_o))))).

Axiom axiom_2a : forall (a : q0_type),

Pf (Implies (Equals (_v (x_a a)) (_v (y_a a)))
(Equals (Apply (_v (h_oa a)) (_v (x_a a)))
(Apply (_v (h_oa a)) (_v (y_a a)))))).

Definition f_ab (a b : q0_type) := V (Fun a b).

Definition g_ab (a b : q0_type) := V_ (Fun a b) 1.

(* Axiom of Extensionality *)

Axiom axiom_3ab : forall (a b : q0_type),

Pf (Equals (Equals (_v (f_ab a b)) (_v (g_ab a b)))
(Forall _ (x_a b)
(Equals (Apply (_v (f_ab a b)) (_v (x_a b)))
(Apply (_v (g_ab a b)) (_v (x_a b)))))).

(* Beta-reduction in \mathcal{Q}_0 *)

Axiom axiom_41v : forall a b : q0_type, forall x : Var a,
forall A : Trm a, forall v: Var b,

diffvar x v ->

Pf (Equals (Apply (Lambda x (_v v)) A) (_v v))).

Axiom axiom_41c : forall a b : q0_type, forall x : Var a,
forall A : Trm a, forall c : Const b,

Pf (Equals (Apply (Lambda x (_c c)) A) (_c c))).

```

Axiom axiom_42 : forall a : q0_type,
    forall x : Var a, forall A : Trm a,
    Pf (Equals (Apply (Lambda x (_v x)) A) A).
Axiom axiom_43 : forall a b g x, forall B : Trm (Fun b g),
    forall C, forall (A : Trm a),
    Pf (Equals (Apply (Lambda x (Apply B C)) A)
        (Apply (Apply (Lambda x B) A)
            (Apply (Lambda x C) A))).
Axiom axiom_44 : forall a g d, forall x : Var a, forall y : Var g,
    forall B : Trm d, forall A,
    diffvar y x ->
    ~ (occurs y A) ->
    Pf (Equals (Apply (Lambda x (Lambda y B)) A)
        (Lambda y (Apply (Lambda x B) A))).
Axiom axiom_45: forall a d, forall x : Var a,
    forall B : Trm d, forall A,
    Pf (Equals (Apply (Lambda x (Lambda x B)) A) (Lambda x B)).

(* Axiom of Descriptions / Axiom of Unique Choice *)
Axiom axiom_5 : forall y,
    Pf (Equals (Apply Iota (Apply (Q I) y)) y).

```

Syntactic Lemmas and Tactics

Axiom projS2_eq :

```
forall (A : Set)(P : A -> Set)(x : A) (p1 p2 : P x),
  existS P x p1 = existS P x p2 -> p1 = p2.
```

Ltac existS_tac H :=

```
let x := type of H in
match x with
  existS ?F ?T ?T1 = existS ?F ?T ?T2 =>
    assert (T1 = T2);
    [apply (projS2_eq q0_type F T T1 T2); assumption | idtac ];
  clear H
```

end.

Ltac rm_existS_tac :=

```
match goal with
| H: existS ?f ?tp ?T = existS ?f ?tp ?T |- _ =>
  clear H; try rm_existS_tac
| H1: ?tp1 = ?tp2,
  H2: existS _ ?tp1 (existS _ _ _) =
    existS _ ?tp2 (existS _ _ _) |- _ =>
  subst tp1; existS_tac H2; try rm_existS_tac
| H: existS _ _ (existS _ _ _) =
  existS _ _ (existS _ _ _) |- _ =>
  existS_tac H; try rm_existS_tac
| H1: ?tp1 = ?tp2, H2: existS _ ?tp1 ?T1 = existS _ ?tp2 ?T2 |- _ =>
  subst tp1; existS_tac H2; subst T1; try rm_existS_tac
| H: existS _ _ ?T1 = existS _ _ ?T2 |- _ =>
  existS_tac H; subst T1; try rm_existS_tac
```

end.

Lemma prefix_p_empty :

```
forall p, prefix p empty -> p = empty.
```

Proof.

```
intro.
```

```

case p; intros.
trivial.
elimtype Logic.False.
inversion H.
elimtype Logic.False.
inversion H.
Qed.

```

```

Lemma disjoint_empty :
  forall p, ~ (disjoint empty p).

```

```

Proof.
  unfold disjoint.
  intros p [H1 H2].
  apply H1; exact (prefix_empty p).
Qed.

```

```

Lemma disjoint_symm :
  forall p1 p2, disjoint p1 p2 -> disjoint p2 p1.

```

```

Proof.
  unfold disjoint; intros p1 p2 [H1 H2].
  auto.
Qed.

```

```

Lemma disjoint_0 :
  forall p1 p2, disjoint (P0 p1) (P0 p2) -> disjoint p1 p2.

```

```

Proof.
  unfold disjoint.
  intros p1 p2 [np1 np2].
  split; intro.
  apply np1.
  exact (prefix_0 p1 p2 H).
  apply np2.
  exact (prefix_0 p2 p1 H).
Qed.

```

```

Lemma disjoint_1 :
  forall p1 p2, disjoint (P1 p1) (P1 p2) -> disjoint p1 p2.
  intros p1 p2 [np1 np2].
  split; intro.
  apply np1.
  exact (prefix_1 p1 p2 H).
  apply np2.
  exact (prefix_1 p2 p1 H).
Qed.

```

```

Lemma replace_empty :
  forall a, forall A B : Trm a, replace A B empty A B.
Proof.
  unfold replace.
  intros; split.
  apply subtrm_empty.
  split.
  apply subtrm_empty.
  intros.
  elimtype Logic.False;
  exact (disjoint_empty p' H).
Qed.

```

```

Lemma replace_apply0 :
  forall a, forall A B : Trm a, forall p,
  forall b c, forall M1 M2 : Trm (Fun b c), forall N : Trm c,
  replace A B p M1 M2 -> replace A B (P0 p) (Apply M1 N) (Apply M2 N).
Proof.
  unfold replace.
  intros a A B p b c M1 M2 N [H1 [H2 H3]].
  split.
  exact (subtrm_apply0 b c a M1 N A p H1).
  split.
  exact (subtrm_apply0 b c a M2 N B p H2).
  intro p1; case p1.

```

```

intro.
elimtype Logic.False.
apply (disjoint_empty (P0 p)).
apply disjoint_symm; assumption.
intros.
assert (disjoint p p0).
apply disjoint_0; assumption.
assert (subtrm Q0 M1 p0 <-> subtrm Q0 M2 p0).
exact (H3 p0 H0 q Q0).
elim H4.
intros.
split; intro; apply subtrm_apply0.
apply H5.
inversion H7.
inversion H8; simpl.
subst b0.
assert (M = M1).
rm_existS_tac; trivial.
rm_existS_tac; assumption.
inversion H7.
rm_existS_tac.
apply H6; assumption.
intros.
split; intro; apply subtrm_apply1;
inversion H0; simpl;
inversion H4; simpl; assumption.
Qed.

```

Lemma replace_apply1 :

```

forall a, forall A B : Trm a, forall p,
forall b c, forall M : Trm (Fun b c), forall N1 N2 : Trm c,
replace A B p N1 N2 ->
replace A B (P1 p) (Apply M N1) (Apply M N2).

```

Proof.

```

unfold replace.

```

```

intros a A B p b c M N1 N2 [H1 [H2 H3]].
split.
apply subterm_apply1; assumption.
split.
apply subterm_apply1; assumption.
intro p1; case p1.
intro.
elimtype Logic.False.
apply (disjoint_empty (P1 p)).
apply disjoint_symm; assumption.
Focus 2.
intros.
assert (disjoint p p0).
apply disjoint_1; assumption.
assert (subterm Q0 N1 p0 <-> subterm Q0 N2 p0).
exact (H3 p0 H0 q Q0).
elim H4.
intros.
split; intro; apply subterm_apply1.
apply H5.
inversion H7; simpl.
inversion H8; simpl.
assumption.
apply H6.
inversion H7; simpl.
inversion H8; simpl.
assumption.
intros.
split; intro; apply subterm_apply0;
inversion H0; inversion H4; simpl.
rm_existS_tac; assumption.
rm_existS_tac; assumption.
Qed.

```

Lemma replace_lambda1 :

```
forall a, forall A B : Trm a, forall p,
forall b c, forall x : Var b, forall N1 N2 : Trm c,
replace A B p N1 N2 ->
replace A B (P1 p) (Lambda x N1) (Lambda x N2).
```

Proof.

```
unfold replace.
intros a A B p b c M N1 N2 [H1 [H2 H3]].
split.
apply subtrm_lambda1; assumption.
split.
apply subtrm_lambda1; assumption.
intro p1; case p1.
intro.
elimtype Logic.False.
apply (disjoint_empty (P1 p)).
apply disjoint_symm; assumption.
Focus 2.
intros.
assert (disjoint p p0).
apply disjoint_1; assumption.
assert (subtrm Q0 N1 p0 <-> subtrm Q0 N2 p0).
exact (H3 p0 H0 q Q0).
elim H4.
intros.
split; intro; apply subtrm_lambda1.
apply H5.
inversion H7; simpl;
inversion H12; simpl;
inversion H14; simpl;
assumption.
apply H6.
inversion H7; simpl.
inversion H12; simpl.
inversion H14; simpl.
assumption.
```

```

intros.
split; intro; apply subtrm_lambda0;
inversion H0;
inversion H4; simpl;
subst b0.
inversion H8; simpl.
inversion H9; simpl.
assumption.
inversion H8; simpl.
inversion H9; simpl; assumption.
Qed.

Ltac replace_tac :=
  match goal with
  | |- replace _ _ empty _ _ =>
    apply replace_empty; replace_tac
  | |- replace _ _ (P0 ?p) (Apply _ _) (Apply _ _) =>
    apply replace_apply0; replace_tac
  | |- replace _ _ (P1 ?p) (Apply _ _) (Apply _ _) =>
    apply replace_apply1; replace_tac
  | |- replace _ _ (P1 ?p) (Lambda _ _) (Lambda _ _) =>
    apply replace_lambda1; replace_tac
  end.

Lemma pos_non_binding_empty :
  forall a, forall A : Trm a, ~ (pos_binding empty A).
Proof.
  repeat intro.
  elim H; intros.
  elim H0; intros.
  elim H1; intros.
  elim H2; intros.
  elim H3; intros.
  elim H4.

```

```

intros.
inversion H5.
Qed.

Lemma pos_non_binding_apply0 :
  forall p s t, forall M : Trm (Fun s t), forall N,
  ~ (pos_binding p M) ->
  ~ (pos_binding (P0 p) (Apply M N)).
Proof.
  unfold pos_binding.
  intros p s t M N H [p2 [s0 [t0 [x [M0 [H1 H2]]]]]].
  apply H.
  inversion H1.
  subst p2.
  elimtype Logic.False.
  inversion H2.
  rewrite H6 in H8; discriminate.
  subst p2.
  exists p1; exists s0; exists t0; exists x; exists M0.
  split.
  assumption.
  inversion H2.
  inversion H3; simpl.
  rm_existS_tac; assumption.
Qed.

Lemma pos_non_binding_apply1 :
  forall p s t, forall M : Trm (Fun s t), forall N,
  ~ (pos_binding p N) ->
  ~ (pos_binding (P1 p) (Apply M N)).
Proof.
  unfold pos_binding.
  intros p s t M N H [p2 [s0 [t0 [x [M0 [H1 H2]]]]]].
  apply H.
  inversion H1.

```

```

subst p2.
exists p1; exists s0; exists t0; exists x; exists M0.
split.
assumption.
inversion H2; simpl.
inversion H3; simpl.
assumption.
Qed.

```

```

Lemma pos_non_binding_lambda1 :
  forall p s t, forall x : Var s, forall N : Trm t,
  ~ (pos_binding p N) ->
  ~ (pos_binding (P1 p) (Lambda x N)).

```

Proof.

```

unfold pos_binding.
intros p s t M N H [p2 [s0 [t0 [x [M0 [H1 H2]]]]]].
apply H.
inversion H1.
subst p2.
exists p1; exists s0; exists t0; exists x; exists M0.
split.
assumption.
inversion H2; simpl.
inversion H8; simpl.
inversion H10; simpl.
assumption.
Qed.

```

```

Ltac pos_non_binding_tac :=
  match goal with
  | |- ~ (pos_binding empty ?A) =>
    apply pos_non_binding_empty
  | |- ~ (pos_binding (P0 ?p) (Apply ?M ?N)) =>
    apply pos_non_binding_apply0; pos_non_binding_tac
  | |- ~ (pos_binding (P1 ?p) (Apply ?M ?N)) =>

```

```

  apply pos_non_binding_apply1; pos_non_binding_tac
| |- ~ (pos_binding (P1 ?p) (Lambda ?x ?N)) =>
  apply pos_non_binding_lambda1; pos_non_binding_tac
end.

```

Lemma pos_binding_apply0 :

```

forall p s t, forall M : Trm (Fun s t), forall N,
pos_binding p M ->
pos_binding (P0 p) (Apply M N) .

```

Proof.

```

unfold pos_binding.
intros.
elim H; intros.
elim H; intros.
elim H1; intros.
elim H2; intros.
elim H3; intros.
elim H4. intros x4 [H5 H6].
exists (P0 x0).
exists x1.
exists x2.
exists x3.
exists x4.
split.
exact (pos_left_0 _ _ H5).
exact (subtrm_apply0 _ _ _ _ _ H6).

```

Qed.

Lemma pos_binding_apply1 :

```

forall p s t, forall M : Trm (Fun s t), forall N,
pos_binding p N ->
pos_binding (P1 p) (Apply M N).

```

Proof.

```

unfold pos_binding.
intros.

```

```

elim H; intros.
elim H0; intros.
elim H1; intros.
elim H2; intros.
elim H3. intros x3 [H4 H5].
exists (P1 x).
exists x0; exists x1; exists x2; exists x3.
split.
exact (pos_left_1 _ _ H4).
exact (subtrm_apply1 _ _ _ _ _ H5).
Qed.

```

```

Lemma pos_binding_lambda1 :
  forall p s t, forall x : Var s, forall M : Trm t,
  pos_binding p M ->
  pos_binding (P1 p) (Lambda x M).

```

```

Proof.
  unfold pos_binding.
  intros.
  elim H; intros.
  elim H0; intros.
  elim H1; intros.
  elim H2; intros.
  elim H3. intros x4 [H4 H5].
  exists (P1 x0).
  exists x1; exists x2; exists x3; exists x4.
  split.
  exact (pos_left_1 _ _ H4).
  exact (subtrm_lambda1 _ _ _ _ _ H5).
Qed.

```

```

Lemma pos_binding_lambda0 :
  forall s t, forall x : Var s, forall M : Trm t,
  pos_binding (P0 empty) (Lambda x M).

```

```

Proof.

```

```

intros.
unfold pos_binding.
exists empty; exists s; exists t; exists x; exists M.
split.
exact pos_left_empty.
exact (subtrm_empty _ (Lambda x M)).
Qed.

```

```

Ltac pos_binding_tac :=
  match goal with
  | |- pos_binding (P0 ?p) (Apply ?M ?N) =>
    apply pos_binding_apply0; pos_binding_tac
  | |- pos_binding (P1 ?p) (Apply ?M ?N) =>
    apply pos_binding_apply1; pos_binding_tac
  | |- pos_binding (P0 ?p) (Lambda ?x ?M) =>
    apply pos_binding_lambda0; pos_binding_tac
  | |- pos_binding (P1 ?p) (Lambda ?x ?M) =>
    apply pos_binding_lambda1
  end.

```

Inference Rules: Rule R and Rule R'

Axiom Rule_R :

```
forall a, forall A B : Trm a, forall p C D,
  Pf C ->
  Pf (Equals A B) ->
  replace A B p C D ->
  ~ pos_binding p C ->
  Pf D.
```

Implicit Arguments Rule_R [a A B C D].

Variable Hyp : Set.

Variable null_Hyp : Hyp.

Variable in_Hyp : Trm 0 -> Hyp -> Prop.

Variable HypPf : Hyp -> Trm 0 -> Set.

Axiom by_assumption: forall A H, in_Hyp A H -> HypPf H A.

Implicit Arguments by_assumption.

Axiom weaken : forall A H, Pf A -> HypPf H A.

Implicit Arguments weaken.

Definition R'_var_condition

```
(a : q0_type)(A B : Trm a)(p : Pos)(C : Trm 0) (H : Hyp) :=
  forall p2,
  prefix p2 p /\ diffpos p2 p -> (*p2 is a proper prefix of p*)
  forall b g, forall x : Var b, forall E : Trm g,
  ~ subtrm (Lambda x E) C p2 \/
  (forall h, in_Hyp h H -> ~ free x h) \/
  ~ free x (Equals A B).
```

Implicit Arguments R'_var_condition.

Lemma R'_var_condition_lem :

```
forall a : q0_type, forall A B : Trm a, forall p : Pos,
  forall C : Trm 0, forall H : Hyp,
  (forall p2,
```

```

prefix p2 p /\ diffpos p2 p ->
forall b g, forall x : Var b, forall E : Trm g,
~ subtrm (Lambda x E) C p2) ->
R'_var_condition A B p C H.

```

Proof.

```

intros.
unfold R'_var_condition.
repeat intro; left.
apply H0.
assumption.

```

Qed.

Axiom Rule_R' :

```

forall H, forall a, forall A B : Trm a, forall C D p,
HypPf H (Equals A B) ->
HypPf H C ->
replace A B p C D ->
~ pos_binding p C ->
R'_var_condition A B p C H ->
HypPf H D.

```

Implicit Arguments Rule_R' [H a A B C D].

Ltac prefix_tac :=

```

match goal with
| H: ~ eqpos _ _ |- _ =>
  elimtype Logic.False; apply H; apply eqpos_empty; try prefix_tac
| |- ~ eqpos _ _ -> prefix _ _ -> _ => intro; intro; try prefix_tac
| H: (?p = empty) |- _ => subst p; try prefix_tac
| H: prefix empty _ |- _ => clear H; try prefix_tac
| H: (_ = _) |- _ => clear H; try prefix_tac
| |- (?p = empty) =>
  apply (prefix_p_empty p); assumption; try prefix_tac
| H: prefix ?p empty |- _ =>
  assert (p = empty); try prefix_tac
| H: prefix (P1 _) (P0 _) |- _ =>

```

```

    elimtype Logic.False; inversion H; try prefix_tac
| H: prefix (P0 _) (P1 _) |- _ =>
    elimtype Logic.False; inversion H; try prefix_tac
| H: prefix (P1 ?p1) (P1 ?p2), H2: ~ eqpos (P1 ?p1) (P1 ?p2) |- _ =>
    inversion H; clear H; assert (~eqpos p1 p2);
    [intro; apply H2; apply eqpos_1; assumption | idtac];
    clear H2; try prefix_tac
| H: prefix (P0 ?p1) (P0 ?p2), H2: ~ eqpos (P0 ?p1) (P0 ?p2) |- _ =>
    inversion H; clear H; assert (~eqpos p1 p2);
    [intro; apply H2; apply eqpos_0; assumption | idtac];
    clear H2; try prefix_tac
| H: prefix ?p _, H2: ~ eqpos ?p _ |- _ =>
    generalize H; generalize H2; case p; clear H H2; try prefix_tac
end.

```

```

Ltac R'_var_cond_tac :=
  match goal with
| p : Pos |- _ => clear p; try R'_var_cond_tac
| |- R'_var_condition _ _ _ _ =>
    unfold R'_var_condition; unfold diffpos; try R'_var_cond_tac
| |- forall p: Pos, prefix _ _ /\ _ _ -> _ =>
    intros _p [_H1 _H2]; try prefix_tac; try R'_var_cond_tac
| |- forall p: Pos, ~eqpos _ _ -> prefix _ _ -> _ =>
    intros _p _H1 _H2; try prefix_tac; try R'_var_cond_tac
| |- forall _ _ _ _, ~ subtrm _ _ empty \/ _ =>
    repeat intro; left; intro _H; inversion _H; try R'_var_cond_tac
| |- forall _ _ _ _, ~ subtrm _ _ (P1 ?p) \/ _ =>
    repeat intro; left; intro _H; try R'_var_cond_tac
| |- forall _ _ _ _, ~ subtrm _ _ (P0 ?p) \/ _ =>
    repeat intro; left; intro _H; try R'_var_cond_tac
| H1: existS _ _ ?A = existS _ _ ?T1,
  H2: existS _ _ ?A = existS _ _ ?T2 |- _ =>
    rewrite H1 in H2; discriminate
| H: subtrm (Lambda _ _) (Apply _ _) empty |- Logic.False =>
    inversion H; try R'_var_cond_tac

```

```
| H:subterm ?A (Apply ?B _) (P0 ?p) |- Logic.False =>
  assert (subterm A B p); inversion H; simpl; try R'_var_cond_tac
| H:subterm ?A (Apply _ ?B) (P1 ?p) |- Logic.False =>
  assert (subterm A B p); inversion H; simpl; try R'_var_cond_tac
| H: existS _ _ _ = existS _ _ _ |- _ =>
  rm_existS_tac; assumption
end.
```

The Meta-Theorems of \mathcal{Q}_0

(* Reflexivity of Equality *)

Theorem Thm5200 :

forall a, forall A : Trm a, Pf (Equals A A).

Proof.

intros.

assert (Pf (Equals (Apply (Lambda (x_a a) (_v (x_a a))) A) A)).

apply axiom_42.

refine (Rule_R (P0 (P1 empty)) H H _ _); unfold Equals.

replace_tac.

pos_non_binding_tac.

Qed.

Implicit Arguments Thm5200.

(* Other basic properties of Equality *)

Theorem Thm5201b :

forall H, forall a, forall A B : Trm a,

HypPf H (Equals A B) ->

HypPf H (Equals B A).

Proof.

intros.

assert (HypPf H (Equals A A)).

apply (weaken H (Thm5200 A)).

refine (Rule_R' (P0 (P1 empty)) H0 H1 _ _ _); unfold Equals.

replace_tac.

pos_non_binding_tac.

R'_var_cond_tac.

Qed.

Theorem Thm5201c :

forall H, forall a, forall A B C : Trm a,

HypPf H (Equals A B) ->

HypPf H (Equals B C) ->

HypPf H (Equals A C).

Proof.

```

intros.
refine (Rule_R' (P1 empty) H1 H0 _ _ _); unfold Equals.
replace_tac.
pos_non_binding_tac.
R'_var_cond_tac.
Qed.

```

Theorem Thm5201d :

```

forall H, forall a b, forall A B : Trm (Fun a b), forall C D : Trm b,
HypPf H (Equals A B) ->
HypPf H (Equals C D) ->
HypPf H (Equals (Apply A C) (Apply B D)).

```

Proof.

```

intros.
assert (HypPf H (Equals (Apply A C) (Apply A C))).
apply (weaken H (Thm5200 (Apply A C))).
assert (HypPf H (Equals (Apply A C) (Apply B C))).
refine (Rule_R' (P1 (P0 empty)) H0 H2 _ _ _); unfold Equals.
replace_tac.
pos_non_binding_tac.
R'_var_cond_tac.
refine (Rule_R' (P1 (P1 empty)) H1 H3 _ _ _); unfold Equals.
replace_tac.
pos_non_binding_tac.
R'_var_cond_tac.
Qed.

```

Theorem Thm5201e :

```

forall H, forall a b, forall A B : Trm (Fun a b), forall C : Trm b,
HypPf H (Equals A B) ->
HypPf H (Equals (Apply A C) (Apply B C)).

```

Proof.

```

intros.
assert (HypPf H (Equals (Apply A C) (Apply A C))).
apply (weaken H (Thm5200 (Apply A C))).

```

```

refine (Rule_R' (P1 (P0 empty)) H0 H1 _ _ _); unfold Equals.
replace_tac.
pos_non_binding_tac.
R'_var_cond_tac.
Qed.

```

Theorem Thm5201f :

```

forall H, forall a b, forall A : Trm(Fun a b), forall C D : Trm b,
HypPf H (Equals C D) ->
HypPf H (Equals (Apply A C) (Apply A D)).

```

Proof.

```

intros.
assert (HypPf H (Equals (Apply A C) (Apply A C))).
apply (weaken H (Thm5200 (Apply A C))).
refine (Rule_R' (P1 (P1 empty)) H0 H1 _ _ _); unfold Equals.
replace_tac.
pos_non_binding_tac.
R'_var_cond_tac.
Qed.

```

Theorem Rule_RR :

```

forall H, forall a, forall A B : Trm a, forall C D p,
Pf (Equals A B) ->
HypPf H C ->
replace A B p C D ->
~ pos_binding p C ->
HypPf H D.

```

Proof.

```

intros.
assert (Pf (Equals C C)).
apply (Thm5200 C).
assert (Pf (Equals C D)).
refine (Rule_R (P1 p) H4 H0 _ _); unfold Equals.
apply replace_apply1; assumption.
apply pos_non_binding_apply1; assumption.

```

```

assert (HypPf H (Equals C D)).
apply (weaken H H5).
refine (Rule_R' empty H6 H1 _ _).
replace_tac.
pos_non_binding_tac.
R'_var_cond_tac.
Qed.

```

Theorem Thm5203 :

```

forall a: q0_type, forall x: Var a, forall A: Trm a,
forall b: q0_type, forall B B': Trm b,
replace_all x A B B' ->
(forall t:q0_type, forall v: Var t, var_in v A -> ~bound_in v B) ->
Pf (Equals (Apply (Lambda x B) A) B').

```

Proof.

Admitted.

Implicit Arguments Thm5203.

Theorem Thm5204 :

```

forall a: q0_type, forall x : Var a, forall A : Trm a,
forall b: q0_type, forall B B' C C': Trm b,
Pf (Equals B C) ->
replace_all x A B B' ->
replace_all x A C C' ->
(forall t : q0_type, forall v : Var t, var_in v A ->
~bound_in v B /\ ~bound_in v C) ->
Pf (Equals B' C').

```

Proof.

```

repeat intro.
assert (Pf (Equals (Apply (Lambda x B) A) (Apply (Lambda x B) A))).
exact (Thm5200 (Apply (Lambda x B) A)).
assert (Pf (Equals (Apply (Lambda x B) A) (Apply (Lambda x C) A))).
refine (Rule_R (P1 (P0 (P1 empty))) H3 H _ _); unfold Equals.
replace_tac.
pos_non_binding_tac.

```

```

assert (Pf (Equals (Apply (Lambda x B) A) B')).
intros.
apply Thm5203.
assumption.
intros.
assert (~bound_in v B /\ ~ bound_in v C).
exact (H2 t v H5).
elim H6.
intros; assumption.
assert (Pf (Equals (Apply (Lambda x C) A) C')).
intros.
apply Thm5203.
assumption.
intros.
assert (~bound_in v B /\ ~ bound_in v C).
exact (H2 t v H6).
elim H7.
intros; assumption.
assert (Pf (Equals B' (Apply (Lambda x C) A))).
refine (Rule_R (P0 (P1 empty)) H4 H5 _ _); unfold Equals.
replace_tac.
pos_non_binding_tac.
refine (Rule_R (P1 empty) H7 H6 _ _); unfold Equals.
replace_tac.
pos_non_binding_tac.
Qed.
Implicit Arguments Thm5204 [a b].

```

References

- [1] TYPES Summer School 2005. *Proofs of Programs and Formalization of Mathematics, Lecture Notes, Volume I*. August 2005.
- [2] Peter B. Andrews. *A Transfinite Type Theory with Type Variables*. North-Holland, 1965.
- [3] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof*. Academic Press, 1986.
- [4] H. P. Barendregt. Introduction to lambda calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg*. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.
- [5] Yves Bertot and Pierre Castéran. Examples and exercises on coq. Technical report, INRIA and LaBRI, 2004. available on www.labri.fr/Persono/sim/casteran/CoqArt/.
- [6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
- [7] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.
- [8] Keith Devlin. Last doubts removed about the proof of the four color theorem. MAA Online, January 2005.
- [9] Keith Devlin. When is a proof? MAA Online, June 2003.
- [10] Christos Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

- [11] Frank Pfenning and Carsten Schürmann. *Twelf User's Guide*, 1.2 edition, September 1998. Available as Technical Report CMU-CS-98-173, Carnegie Mellon University.
- [12] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.
- [13] L.-Y. Tan. Formalizing the Meta-Theory of \mathcal{Q}_0 in Rogue-Sigma-Pi. In Judit Germain, editor, *17th European Summer School in Logic, Language and Information*, 2005.
- [14] LogiCal Project The Coq Development Team. The Coq Proof Assistant Reference Manual, Version 8.0. Available online at <http://coq.inria.fr/doc/main.html>.

Vita

Li-Yang Tan

Date of Birth	October 12, 1985
Place of Birth	Republic of Singapore
Degrees	B.A. Mathematics, expected May 2006
Honors	Elected Phi Beta Kappa, Sigma Xi Washington University Dean's List, 4 semesters
Professional Societies	Mathematical Association of America (MAA)
Publications	<p>Tan, Li-Yang (2006). The Chromatic Symmetric Function of Trees, <i>Senior Thesis, Mathematics</i>. Available online at http://www.cs.wustl.edu/~lt1.</p> <p>Tan, Li-Yang (2006). Combinatorial Calculations in the Free Tree Quotient Modules of the Grossman-Larson Hopf Algebra, <i>Senior Thesis, Mathematics</i>. Available online at http://www.cs.wustl.edu/~lt1.</p> <p>Tan, Li-Yang (2005). Formalizing the Meta-Theory of \mathcal{Q}_0 in Rogue-Sigma-Pi, <i>Proceedings of the 16th European Summer School in Logic, Language and Information</i>.</p> <p>Stump, Aaron and Tan, Li-Yang (2005). The Algebra of Equality Proofs, <i>Proceedings of the 17th International Conference on Rewriting Techniques and Applications</i>.</p>

May 2006

Short Title: \mathcal{Q}_0 in Coq

Tan, M.Sc. 2006