
Formalizing the Meta-Theory of \mathcal{Q}_0 in Rogue-Sigma-Pi

LI-YANG TAN

Washington University in St. Louis, Computational Logic Group

lytan@wustl.edu

ABSTRACT. Introduced by Peter Andrews in the 1960's, \mathcal{Q}_0 is a classical higher-order logic based on simply-typed lambda calculus. This paper presents our work in progress on the formalizing of \mathcal{Q}_0 in a programming language, Rogue-Sigma-Pi (RSP), with the aim of validating its meta-theory. The main challenge of this project arises from the fact that while all logical derivations are carried out in much detail in Andrews' formalism, many of the syntactic derivations have been kept implicit. Therefore, most of our work has been devoted to setting up a framework that allows us to formalize low-level syntactic notions of \mathcal{Q}_0 , such as variable occurrences, bindings and replacement. This formalization also includes proving meta-theoretic properties of these various syntactic notions. Building on the the ability to prove syntactic derivations assumed in Andrews' formalism, recent progress has led to the proving of basic meta-theorems of \mathcal{Q}_0 , such as the equality rules, alpha-equivalence, beta- and eta-conversions, as well as capture-avoiding substitution. This paper will discuss the theoretical and engineering challenges faced in our formalizing of \mathcal{Q}_0 in RSP that is guided by a faithful adherence to Andrews' presentation on paper.

1 Introduction

In studying Peter Andrews' formalism of \mathcal{Q}_0 [1], one notices that all logical derivations are made very explicit and carried out meticulously. In fact, one is often 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: 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} . Although such syntactic lemmas are arguably intuitive and straightforward (and hence

kept implicit in Andrews’ formalism), our meta-language is not expressive enough to prove the associated syntactic derivations. \mathcal{Q}_0 ’s complex syntactic notions pose additional challenges. Type theoretic meta-languages like RSP are commonly used for formalizing meta-theories, as they often have support for customary syntactic notions such as capture-avoiding substitution for all occurrences of a variable. However, \mathcal{Q}_0 is based on more complex syntactic notions — for instance, substitution in \mathcal{Q}_0 allows for variable capture sometimes, and only replaces a single occurrence.

Hence, we need to set up a framework within which we can formalize these syntactic notions. With the added expressiveness of our meta-language, we can then state and prove the syntactic lemmas needed for our meta-proofs. A judgmental formulation seems appealing and was the approach we choose to take initially. However, there are too many related syntactic notions (see Table 1.1), and formalizing them all independently proved to be unfeasible. Therefore, we introduced a *sublogic* — an underlying logical system of our formalism. Our sublogic is a multi-sorted classical first-order natural deduction system. Every syntactic notion is thus defined by a first-order formula, and syntactic derivations correspond to proofs in our natural deduction system. This is the main contribution of our project and will be the emphasis of this paper.

2 The Primitive Basis of \mathcal{Q}_0 in RSP

2.1 Rogue-Sigma-Pi

Rogue-Sigma-Pi (RSP) is a functional language designed for the manipulation of proofs [8], and it includes the Edinburgh Logical Framework (LF) [6] as proper fragment. Based on dependent type theory, types in RSP are indexed by terms and programs manipulate dependently typed data. For this project, RSP serves the same functionality as theorem provers such as Isabelle, Nuprl, Coq, etc [7, 4, 5]. The basic concrete syntax of RSP is given below.

c	$::$	A	<i>type ascription</i>
$x:A$	\Rightarrow	B	<i>dependent function type</i>
A	\Rightarrow	B	<i>non-dependent function type</i>
$x:A$	\rightarrow	M	<i>lambda abstraction</i>

A key aspect of our approach is the use of tactics to build proofs of various syntactic lemmas. In our current framework, we have a soundness guarantee for our theorem prover (RSP): if a tactic is well-typed, then any proof it

builds is guaranteed to be valid. However, we have no such guarantee for completeness. Our group is currently working on RSP1 [9], an extension of RSP with simple termination and coverage checking. The type theory of RSP1 will allow us to provide a total correctness guarantee for our tactics. We will then be able to statically validate the meta-theory of \mathcal{Q}_0 . However, as RSP1 is still work in progress, our tactics are only partially verified in our current framework; certain bugs can only be detected when the RSP programs are executed.

Higher-order abstract syntax (HOAS) has been advocated as a way of avoiding the overhead of explicit reasoning about syntax by incorporating the necessary syntactic properties into the meta-logic. In fact, RSP does support HOAS. However, due to the peculiar nature of \mathcal{Q}_0 's syntactic notions, HOAS is not a suitable choice for this project. For example, as mentioned in the previous section, substitutions allowed by replacement rules are linear, and variable-capturing in some cases.

2.2 \mathcal{Q}_0 types, variables, constants and terms

In this section, we will discuss the embedding of the primitive basis of \mathcal{Q}_0 in RSP. A presentation of \mathcal{Q}_0 types, variables, constants and terms will run parallel with a discussion of their corresponding representations in RSP.

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

$$\begin{array}{ll} \text{types } t ::= & i \qquad \text{type of individuals} \\ & | \ o \qquad \text{type of truth values} \\ & | \ (\alpha\beta) \qquad \text{type of functions from elements} \\ & \qquad \qquad \qquad \text{of type } \beta \text{ to elements of type } \alpha \end{array}$$

With the exception of the abstraction operator λ and parentheses $[]$, the only primitive symbols of \mathcal{Q}_0 are variables and logical constants. \mathcal{Q}_0 variables in RSP are indexed by their types and also by a variable number. There are two logical constants: $\mathbf{Q}_{((o\alpha)\alpha)}$ and $\iota_{(i(o))}$. The former denotes the identity relation between elements of type α and the latter, a description operator for individuals. The typing of \mathcal{Q}_0 types, variables and constants in RSP are given below. Note that **type** is a primitive *kind* of RSP.

```

Tp :: type.
0  :: Tp.
I  :: Tp.
Fun :: Tp => Tp => Tp.
Var :: Tp => type.

```

```

Const :: Tp => type.

V :: a : Tp => V_num => Var a.           # variables
Q :: s : Tp => Const (Fun (Fun 0 s) s). # identity relation
Iota :: Const (Fun I (Fun 0 I)).        # description operator

```

In \mathcal{Q}_0 , well-formed formulas and their corresponding types (denoted by uppercase letters subscripted by their types—e.g., \mathbf{A}_α , \mathbf{B}_β , $\mathbf{C}_{\alpha\beta}$, ...) are recursively defined as follows. We abbreviate well-formed formulas of type α by wff_α .

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.

Well-formed formulas are terms in our formalization and they have type $\text{Trm } X$, where X is a \mathcal{Q}_0 type (with RSP type Tp). The typing of each of the four possible forms of terms is given below.

```

Trm :: Tp => type.
_v :: s : Tp => Var s => Trm s.         # injection of a variable
_c :: s : Tp => Const s => Trm s.       # injection of a constant
Apply :: a : Tp => b : Tp => Trm (Fun a b) => Trm b => Trm a.
Lambda :: a : Tp => b : Tp => Var b => Trm a => Trm (Fun a b).

```

Before stating the axioms and inference rule of \mathcal{Q}_0 , Andrews introduces several definitions of the customary logical operators ($\wedge, \vee, \supset, \sim, T, F, \dots$) and their corresponding syntactic abbreviations. These have all been encoded in RSP but due to space considerations, they will not be replicated here.

2.3 Axioms of \mathcal{Q}_0 and Rule R

\mathcal{Q}_0 has five axioms, two of which ($3^{\alpha\beta}$ and $4_1 - 4_5$) are axiom schemata. Collectively, they describe basic properties about truth, falsehood, equality and λ . Also, the \mathcal{Q}_0 equivalents of the Axiom of Extensionality and the Axiom of Descriptions are both stated. As with the definitions and abbreviations, all five axioms have been encoded in our formalism in RSP.

\mathcal{Q}_0 's single rule of inference is a rule of replacement which allows for variable capture. (Note: $\mathbf{A}, \mathbf{B}, \mathbf{C}$ etc. are abbreviations for the wff_o 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 .

There are three syntactic notions central to this rule, all of which have to be formalized in RSP before we can state Rule R. First, we have to define when a \mathcal{Q}_0 term occurs in another. That is, we have to formalize the ternary relation “ \mathbf{A} is a subterm of \mathbf{B} at position p .” Next, we need to formalize the notion of replacing one term by another at a certain position. Third, we need to define when a position is binding, and non-binding, in a term.

This necessitates an underlying logical system that will allow us to formalize such syntactic relations among \mathcal{Q}_0 terms and variables, and then allow us to build proofs of meta-theoretic properties resulting from these relations. In this way, we are able to explicitly carry out the syntactic derivations that are assumed in Andrews’ meta-proofs. The resulting architecture of our approach can be summarized as follows: We encode our sublogic within the logical framework RSP. On top of this sublogic, we define \mathcal{Q}_0 . The incorporation of the sublogic into our formalism will be described in detail in the following sections.

3 Formalizing \mathcal{Q}_0 ’s Syntactic Notions with a Sublogic

Our sublogic 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. As proofs of more elaborate syntactic notions can be of considerable size, we also implement validated tactics to alleviate the burden of building proofs by hand.

The semantics of the name “sublogic” should be clarified here. In mathematics, the prefix “sub” is commonly understood as follows: sub- X is a part of X that inherits all the structure of X . A few examples are subgroup, subgraph, and subspace. However, in this paper, our sublogic is an auxiliary

logic upon which \mathcal{Q}_0 is defined. Therefore, we do not mean it in the same sense as how classical propositional logic is a sublogic of first-order predicate logic, but instead, our sublogic is an underlying logical system of \mathcal{Q}_0 .

3.1 Defining Syntactic Notions in the Sublogic

In our sublogic, we take equality between sublogic terms of the same sort as primitive syntactic notions. In addition, the subterm, prefix, left child, right child and set membership relations are taken primitive. All other syntactic notions are defined in terms of the above primitive notions. Table 1.1 lists the syntax and semantics of all primitive syntactic notions, along with some of the basic derived notions of our sublogic.

Given the primitive and basic derived notions listed in Table 1.1, our sublogic is now sufficiently expressive to formalize some of \mathcal{Q}_0 's more elaborate syntactic notions. We will illustrate this with an example: Given terms \mathbf{A}_α and \mathbf{B}_β and a variable x_α , \mathbf{A}_α is *free for* x_α in \mathbf{B}_β if and only if no free occurrence of x_α in \mathbf{B}_β is in a well-formed part of \mathbf{B}_β of the form $[\lambda y_\gamma \mathbf{C}_\delta]$ where y_γ is free in \mathbf{A}_α . Equivalently, if x_α occurs free in a well-formed part of \mathbf{B}_β of the form $[\lambda y_\gamma \mathbf{C}_\delta]$, then y_γ is not free in \mathbf{A}_α . The typing and definition of *free for* in our sublogic is shown below.

```

free_for ::
  a : Tp => b : Tp => Trm a => Var a => Trm b => sl_o =
  a : Tp -> b : Tp -> A : Trm a -> x : Var a -> B : Trm b ->
  for_all_pos (p : Pos ->
  for_all_tp (gamma : Tp ->
  for_all_tp (delta : Tp ->
  for_all_var (y : Var gamma ->
  for_all_trm (C : Trm delta ->
  sl_imp (sl_and (sl_subtrm Lambda(y,C) B p)
            (sl_free_var x Lambda(y,C))
            (sl_not_free y A)))))).

```

3.2 Sublogic Proofs of Syntactic Notions: Lemmas and Tactics

For some x_α , \mathbf{A}_α and \mathbf{B}_β , we prove that \mathbf{A}_α is free for x_α in \mathbf{B}_β by doing a case analysis on \mathbf{B}_β . If $\mathbf{B}_\beta \equiv v_\beta$ for some variable v_β , the claim holds true vacuously since there cannot be a lambda-term in v_β . On the other hand, if $\mathbf{B}_\beta \equiv [\mathbf{M}_{\beta\eta}\mathbf{N}_\eta]$, it is necessary and sufficient to establish that \mathbf{A}_α is free for x_α in $\mathbf{M}_{\beta\eta}$, and in \mathbf{N}_η . The other two cases for constants and lambda

abstractions are similar.

In our formalism, we first state and prove four sublogic lemmas, one for each possible form of \mathbf{B}_β . The proofs of these lemmas are derivations within our natural deduction system. For example, in the case of $\mathbf{B}_\beta \equiv v_\beta$, the implication is established by first assuming the antecedent, obtaining the proof that $[\lambda y_\gamma \mathbf{C}_\delta]$ occurs in v_β using $\wedge E_L$, and then deriving the consequent via *ex falso quolibet*. If \mathbf{B}_β is an application or a lambda abstraction, the proof is a hypothetical derivation from assumption(s) that \mathbf{A}_α is free for x_α in certain subterm(s) of \mathbf{B}_β . For example, for the case $\mathbf{B}_\beta \equiv [\mathbf{M}_{\beta\eta} \mathbf{N}_\eta]$, the corresponding lemma has to take in a sublogic proof that \mathbf{A}_α is free for x_α in $\mathbf{M}_{\beta\eta}$ (that is, an RSP term with type `sl_pf (free_for a (Fun b e) A x M)`), and similarly for \mathbf{N}_η . The typing of `free_for_apply` is shown below.

```

free_for_apply ::
  a : Tp => b : Tp => e : Tp => Trm a => Var a =>
  M : Trm (Fun b e) => N : Trm e =>
  sl_pf (free_for a (Fun b e) A x M) =>
  sl_pf (free_for a e A x N) =>
  sl_pf (free_for a b A x Apply(M,N)).

```

For each syntactic relation, we implement a tactic that, given arbitrary input terms, builds the corresponding sublogic proof that the relation holds among them (or returns `Null` if it does not). All our syntactic lemmas have been proven using case analysis, and case analysis is carried out with pattern matching in RSP (syntactic first-order matching). Hence, in `free_for`, each of the four possible forms of \mathbf{B}_β is represented by a dependently typed pattern abstraction, and the abstractions are combined with deterministic choice. When a pattern matches the target, our tactic calls the corresponding lemma for that form. For the lemmas that are hypothetical derivations (e.g., `free_for_apply`), recursive calls yield the desired proofs of assumptions.

Numerous other syntactic relations and lemmas have been proven in our formalism. A particularly challenging syntactic notion arises in the statement of a generalization of Rule R. Replacement in Rule R' is subject to the following restriction: 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 $[\mathbf{A}_\alpha = \mathbf{B}_\alpha]$ (\mathcal{H} is a list of hypotheses). The formalizing of this syntactic restriction in our sublogic — statement of lemmas, their associated sublogic proofs, along with tactics — entailed more than 1200 lines of RSP code.

Having stated and proven the necessary syntactic lemmas, we have recently started proving elementary meta-theorems of \mathcal{Q}_0 . Equality rules, alpha-equivalence, restricted and generalized versions of beta-conversion, eta-conversion, as well as a generalization of Rule R', have all been stated, proven and verified in our formalism. With the added expressiveness of our framework, the programs written to prove these meta-theorems have all been relatively small — the total size of the first seven meta-theorems is only slightly more than 1300 lines of RSP code. The size of the \mathcal{Q}_0 project currently stands at 11000 lines of RSP code.

4 Conclusion and Future Work

In this paper, we have discussed the challenges involved in the formalizing of Peter Andrews' classical higher-order logic \mathcal{Q}_0 in our language Rogue-Sigma-Pi. The main challenge of this project has been the setting up of a framework within which we can define the complex syntactic notions of \mathcal{Q}_0 precisely. Our proposed solution is to embed an underlying logical system — a first-order natural deduction system — into our formalism. This sublogic provides us with the expressiveness to first define, and then prove meta-theoretic properties about the various syntactic notions. We are then able to explicitly carry out syntactic derivations assumed in Andrews' formalism and prove syntactic lemmas needed for the meta-proofs of \mathcal{Q}_0 .

Having completed this phase of the project, we have since gone on to state and prove some of the elementary meta-theorems of \mathcal{Q}_0 . Although the syntactic theory we have stated in the sublogic may need further additions or even revisions as we proceed, the proving of meta-theorems is now the main emphasis of the project. It is our hope that the project will culminate in the proving of the Deduction Theorem of \mathcal{Q}_0 .

5 Acknowledgments

First and foremost, I would like to thank my advisor, Professor Aaron Stump, for his guidance throughout this project. Also, I would like to thank the anonymous reviewers for their helpful comments and suggestions. Last but not least, I would like to thank the members of my research group, Ian Wehrman, Edwin Westbrook, Robert Klapper, and Joel Brandt, for their support.

Primitive Sublogic Syntactic Notions	
<code>sl_eq_tp a b</code> <code>sl_eq_var v1 v2</code> <code>sl_eq_const c1 c2</code> <code>sl_eq_trm A B</code> <code>sl_eq_pos p1 p2</code> <code>sl_eq_vnum N M</code>	Holds iff two sublogic terms of the same sort are syntactically equal.
<code>sl_subtrm A B p</code>	A is a subterm of B at position p
<code>sl_prefix p1 p2</code>	Position p_1 is a prefix of p_2
<code>sl_pos_left p1 p2</code>	$p_2 = p_1 \cdot 0$. That is, p_2 is the result of going left on p_1 .
<code>sl_pos_right p1 p2</code>	$p_2 = p_1 \cdot 1$
<code>sl_olist_member x L</code>	x is in the list of hypotheses L
Some Derived Sublogic Syntactic Notions	
Inequality	
<code>sl_diff_tp a b</code> <code>sl_diff_var v1 v2</code> <code>sl_diff_const c1 c2</code> <code>sl_diff_trm A B</code>	Holds iff the two \mathcal{Q}_0 terms of the same sublogic sort are syntactically different. Negation of <code>sl_eq_*</code>
Positions	
<code>sl_pos_binding p A</code>	Position p in A is immediately preceded by a λ
<code>sl_pos_non_binding p A</code>	Negation of <code>sl_pos_binding</code>
Variable Occurrences and Bindings	
<code>sl_is_var v1 A</code>	The variable v_1 occurs in A .
<code>sl_not_var v1 A</code>	Negation of <code>sl_is_var</code>
<code>sl_bound_above v A p</code>	Position p in A is bound above by v .
<code>sl_not_bound_above v A p</code>	Negation of <code>sl_bound_above</code>
<code>sl_free_var v A</code>	v occurs free in A
<code>sl_not_free v A</code>	Negation of <code>sl_free_var</code>
<code>sl_bound_var v A</code>	v occurs bound in A .
<code>sl_not_bound v A</code>	Negation of <code>sl_bound_var</code>
Replacement	
<code>sl_replace x y p C D</code>	Replacing x by y at position p in C gives you D
<code>sl_replace_all x A C D</code>	Replacing all free occurrences of x by A in C gives you D

Table 1.1: Syntactic Notions defined in the Sublogic

Bibliography

- [1] P. Andrews. *A Transfinite Type Theory with Type Variables*. North-Holland, 1965.
- [2] P. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof*. Academic Press, 1986.
- [3] A. Appel and A. Felty. Dependent Types Ensure Partial Correctness of Theorem Provers. *Journal of Functional Programming*, 14(1):3–19, January 2004.
- [4] R. Constable and the PRL group. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, 1986.
- [5] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.
- [6] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
- [7] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
- [8] A. Stump. Imperative LF Meta-Programming. In C. Schürmann, editor, *4th International Workshop on Logical Frameworks and Meta-Languages*, 2004.
- [9] E. Westbrook and A. Stump. RSP1: A First-Order, Dependently Typed Pattern Matching Language with Imperative Features, 2005. Manuscript available at <http://cl.cse.wustl.edu/papers/rsp1.pdf>.