A CONSTRUCTIVE SEMANTICS FOR REWRITING LOGIC

BY

MICHAEL N. KAPLAN

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

COMPUTER SCIENCE

UNIVERSITY OF RHODE ISLAND

2015

DOCTOR OF PHILOSOPHY DISSERTATION

OF

MICHAEL N. KAPLAN

APPROVED:

Dissertation Committee:

Major Professor Lutz Hamel

Lisa DiPippo

Lubos Thoma

Nasser H. Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2015

ABSTRACT

We present an alternative model theory for rewriting logic in the calculus of inductive constructions. Making use of the Coq theorem proving environment, we create a formal specification of rewriting logic, and gain an interactive logical and semantic framework for rewriting logic, which is a logic well suited to the specification and validation of concurrent computations and proofs. We then show how we can use Coq and this specification as a formal proof environment for working with rewriting logic theories. Finally, we present tactics to automate portions of the proof generation and validation process.

ACKNOWLEDGMENTS

Thanks to Lorraine Berube, secretary of the Computer Science and Statistics department, for her never-ending cheer and helpfulness.

And finally, thanks to my academic advisor, Lutz Hamel, for his tutelage, patience and counsel during my extended years in his care.

DEDICATION

I dedicate this thesis to my wife Karin, and my children, Mika and Allen. It's been quite the journey. Thanks.

They have all made tremendous sacrifices to allow me to complete these studies. No words can adequately express my gratitude.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGMENTS iii

DEDICATION iv

TABLE OF CONTENTS v

CHAPTER

1 Introduction 1

1.1 Overview 1

1.2 Problem Discussion 3

1.2.1 Formalization of Rewriting Logic 3

1.2.2 Theorem Proving in Rewriting Logic 4

1.3 Contribution 5

1.4 Related Work 5

1.4.1 Full Maude and Rewriting Logic Tools 6

1.4.2 Coq Formalizations of Term Rewriting 8

1.4.3 Equational Logic in Type Theory 9

1.4.4 Coq Formalizations of Logic 9

1.4.5 Coq Formalizations of Concurrency 10

1.5 Structure of Thesis 11

2 Background 12

2.1 Term Rewriting 12

2.1.1 Syntax and Semantics 14

Page

2.1.2 Properties of Term Rewriting Theories 14
2.2 Formal Logic 16
2.3 Universal Algebra 19
2.4 Calculus of Inductive Constructions 20
3 Rewriting Logic 25
3.1 Syntax 25
3.2 Semantics 29
3.2.1 Proof Theory 29
3.2.2 Model Theory 34
3.3 Soundness and Completeness 41
3.3.1 Soundness 41
3.3.2 Completeness 46
3.3.3 Sound and Complete 49
4 Example 50
5 Theorem Proving in Rewriting Logic 55
5.1 Tactics 55
6 Conclusions and Future Work 63
6.1 Future Work 63
6.1.1 Equational Sublogic 63
6.1.2 Verified Extracted Implementation 65
6.1.3 Quotient Relationships 66
6.1.4 Type System 66
6.2 Conclusion 67

Page

LIST OF REFERENCES 68

APPENDIX

Additional Source Code 71

A.1 Utilities 71

A.2 Tactic Utilities 76

A.3 Equivalence Proof 78

BIBLIOGRAPHY 81

CHAPTER 1

Introduction

1.1 Overview

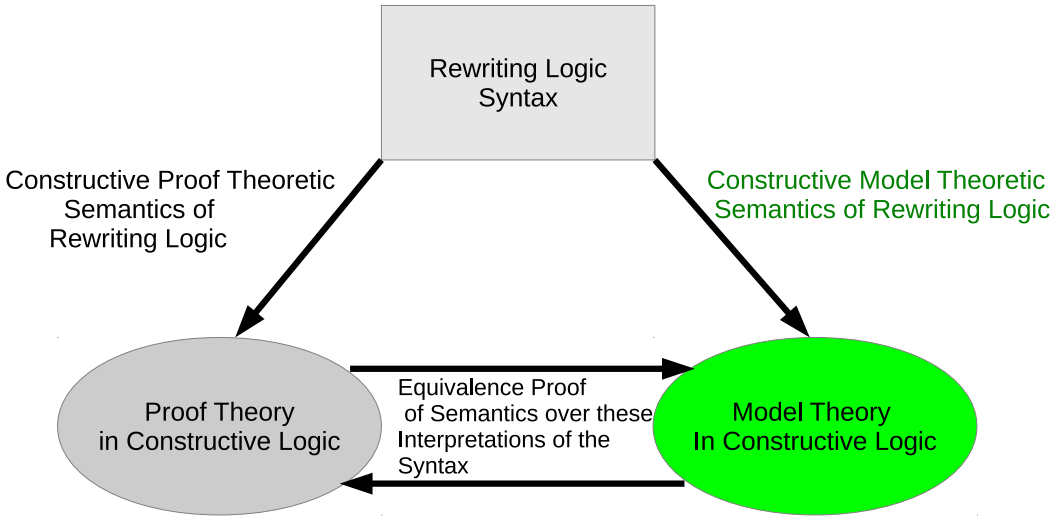
This work develops the theory of rewriting logic by creating a constructive semantic model in the calculus of inductive constructions as well as a constructive proof-theoretic deduction system based on the proof theory originally defined by Meseguer [1]. These two semantics are proven equivalent to demonstrate that rewriting logic is both sound and complete.

A sound logic is one for which every sentence which can be derived in the proof theory is also satisfiable in the model theory. This is the same as saying that if a sentence is derivable, it must be true, which is a fairly essential property for a logic. Being able to derive a false sentence would certainly limit any confidence in the logic.

A complete logic is one for which every sentence satisfiable in the model theory corresponds to a deductive proof that can be constructed in the proof theory. In other words a logic is complete if every true sentence is also derivable.

Additionally, it is shown how these semantics can be used as a theorem proving environment for working with rewriting logic theories making use of the Coq [2] theorem proving environment.

The impetus for this work is the goal of creating secure, reliable and robust software and systems, specifically specifying and validating concurrent systems. As computers continue to become more complex as multi-core and distributed system usage increases making the creation of efficient and useful software increasingly



difficult. At the same time, our reliance on software in critical systems including infrastructure, avionics, and banking continues growing at a blistering pace. As the costs for errors and failures are increasing in time with the difficulty of creating robust systems, adopting new development techniques becomes more appealing and cost feasible.

To help address the need for more robust concurrent software, this research seeks to advance rewriting logic, which is a formal system suitable for this specification and validation. In brief, rewriting logic expresses a system via a set of directed-rules which operate by modifying a term which we can think of as the state of a system. Rewriting logic therefore functions similarly to a state-transition machine, and it allows multiple simultaneous transitions, allowing us to reason over specifications

of concurrent systems. A new model will allow for multiple interpretations of a specification to accompany the more fixed proof-theoretic view as we discuss in subsection 3.2.2 and subsection 6.1.3.

1.2 Problem Discussion

1.2.1 Formalization of Rewriting Logic

We begin our journey into the formalization of concurrent and distributed software by first creating a new formalization of rewriting logic in the calculus of inductive constructions. Our goal is to create versions of a model theory and proof theory that be proven equivalent in a machine verifiable fashion verifying that the new model theory captures the desired semantics from the existing proof theory. We can call this a *verified semantics*.

As described in greater detail in Section 2.2, to formalize a logic we need to define its syntax and semantics, where the semantics really consists of both a deductive system, the proof theory, and a mathematical model, the model theory. We must then detail the relationship between them, considering the properties of soundness and completeness.

Our first step towards our verified semantics is to define the syntax of rewriting logic, in Section 3.1, which we will then use as the basis for the later semantics. Our goal with the specification of the syntax, and indeed the entire system, is not currently for ease of use, therefore the syntax is in a very raw form. Prior to defining our syntax, we will use an alternative syntax for rewriting logic from the rewriting logic language Maude [3] in our examples.

1.2.2 Theorem Proving in Rewriting Logic

In rewriting logic, we would like to provide specifications of potentially concurrent systems, and have the means to deduce information from the specification. Many systems implementing deduction in rewriting logic exist, including ELAN [4], CafeOBJ [5], and Maude. We describe a simple rewrite theory in the syntax of Maude below specifying peano numbers along with a non-deterministic choice operation.

```
mod NAT-CHOICE is
 sorts A .
 op 0 : → A .
 op S _ : A → A .
 op ? _ _ : A A → A .
 vars V1 V2 : A .
 rl [left] : ? V1 V2 ⇒ V1 .
 rl [right] : ? V1 V2 ⇒ V2 .
endm
```

In the above specification, we define a theory with the name *NAT-CHOICE*. This specification contains a single sort, or type, "A", used in defining the operations and rules. Our treatment of rewriting logic is untyped, so we simulate this in our Maude examples by using a single sort. *NAT-CHOICE* consists of three operations, "0", "S", and "?" of arities 0, 1 and 2 respectively. These arities are shown above via the number of underscores following the operation declaration. The operations also have type signatures which also represent the arity in the number of parameters. The type signature for *?* is of the form $A\ A \rightarrow A$ indicating that the *?* operator takes two elements, and returns one element.

We then define two rules over these operations, labeled "left" and "right", which reduce to either the left or right operands respectively. Maude supplies a *rewrite* command which is used to perform deduction. *Rewrite* takes a theory and a

ground term and produces a new ground term derivable in the theory from the initial ground term where a *ground term* is a term which contains no variables. In the below example, this command should non-deterministically return either "O" or "S(O)" as a result based on the proof theory (see Subsection 3.2.1) of rewriting logic, as we see in the deduction trace below.

```
> rewrite in NAT-CHOICE: choose 0 S(0) .
rewrites: 1 in 0ms cpu (0ms real) (  rewrites/second)
result S: 0
```

Here Maude has found that the ground term "O" is derivable in one rewrite step.

1.3 Contribution

The major contribution of this thesis is the introduction of an alternative model theoretic semantics for rewriting logic in Subsection 3.2.2. We show in Section 3.3 that with this semantics, rewriting logic is sound and complete with respect to the original proof theoretic semantics defined by the original author [1]. Additionally, the constructive proof theoretic semantics defined in Subsection 3.2.1 allows for machine verifiable deduction in rewriting logic explored further in Chapter 5.

1.4 Related Work

There exists a large existing body of work on rewriting logic, on the formalization of concurrent systems, and on the formalization of concurrent systems. We will discuss a few of the relevant points below.

1.4.1 Full Maude and Rewriting Logic Tools

The Full Maude implementation of rewriting logic is a machine executable specification that has been used to experiment with extensions to the Maude language [6]. This implementation was used to integrate object-oriented programming with rewriting logic, to add parametrized modules, views and module expressions. These are all purely syntactic constructs to make Maude, the programming language, more appealing to work in, but do not change the underlying semantics of rewriting logic.

In a similar vein, the logical foundations of ELAN are formalized in rewriting logic in [7], where the emphasis is on the notion of strategies for computing rewrites. Strategies allow us to provide a search path by specifying which reductions occur when. For instance, we can specify that rewriting should proceed in a left-most, inner-most fashion such the top level term is never reduced until all of subterms can no longer be reduced. These strategies are an implementation detail for executing theories but do not change the theory behind the logic.

Another meta-level tool for working with rewrite theories is the Church-Rosser Checker [8]. For Maude functional modules to function as executable specifications, it is a requirement that they be Church-Rosser and terminating. This guarantees that all chains of equational simplification lead to a canonical form modulo the equational attributes. Maude itself makes no attempt to verify that a module meets these requirements. In fact, Maude will generally accept functional modules that don't meet the above and attempt to execute them leading to generally undesired behavior. The Church-Rosser tool provides a way for users to validate their theories before attempting to run them in Maude.

The Coherence Checker [9] is similar to the Church-Rosser checker, but checks that system modules are coherent with respect to their equational rules modulo the equational attributes. To check coherence, the appropriate critical pairs between the rewrite rules and equations must be able to be filled in. This allows intermingling of performing rewrites of rules and rewrites of equations without losing completeness. The coherence checker only works when the theory is both terminating and Church-Rosser. The Maude interpreter itself is again unable to automatically check this property.

There is an effort address the limitations in Maude by integrating all of these separate tools into a unified environment called the Maude Formal Environment [10]. Since many of these tools rely on the supplied theory having certain properties which can be checked by other tools, the integrated environment creates a more trusted platform with which to run the validations. The Maude Foral Environment creates an environment where the tool results and proofs are objects that can be shared. This allows, for example, the coherence checker to access the results from the Church-Rosser and termination checkers to be able to state that a theory is confluent, without any qualifications.

My work seeks to encourage similar exploration of rewriting logic extensions and tools, but to provide a different avenue for reasoning about rewriting logic. A potential idea would be to specify Church-Rosser (or Coherence) as a type and then provide tools to help prove that an equational specification inhabits this type. In this way, my rewriting logic implementation could automatically reject incomplete or incorrect specifications in the type-checking phase instead of requiring an external tool.

1.4.2 Coq Formalizations of Term Rewriting

There exists an ongoing project, CoLoR [11], for creating a Coq library on rewriting with a focus on termination. The focus is in the form of the formalization of various termination checking techniques. The goal is to formalize the techniques most commonly used in automated termination provers, so that the termination proof certificates that are generated by the automated termination tools can be verified in Coq. The verification is done via the creation of a Coq file corresponding to the termination problem used in the proof certificate as well a formalization of the parameters in the certificate and an encoding of the expected termination result.

As rewriting logic is implicit in term rewriting, and termination is a necessary property of the functional modules in Maude, reuse of some of the formalizations may be possible. The CoLoR library also contains some basic math and data structure theories in Coq which have already proven general enough to be used in other Coq formalizations by outside groups. There is a wealth of basic libraries both part of the basic Coq distribution, and in the form of user-contributed modules that help to make Coq an attractive environment for this research.

Other formalizations of models of concurrency have been conducted in Coq including a higher-order abstract syntax based encoding of the pi-calculus [12], and a specification of the theories of process algebra, in particular the Calculus of Communication Systems, CCS. As these are alternative models to rewriting logic, and can also be specified within rewriting logic, ideas from both of these specifications should prove beneficial, particularly the use of Co-induction in the specification of the pi-calculus.

1.4.3 Equational Logic in Type Theory

Rewriting logic and equational logic are closely linked, and many implementations of rewriting logic emphasize the existence and use of an equational sublogic to describe static portions of a theory. The models of equational logic are therefore of particular interest for future extensions of our model of rewriting logic.

Previous formalizations of equational logic in Coq [13] followed the use of quotient algebras [14, 15] as a model for equational logic. First we define a term algebra over the signature of an equational theory, where the objects are the generated terms, with variables, and the methods are the generator functions. Then a quotienting relation can be defined where a pair of terms is considered equal if it is derivable in the equational theory. This work defines a Coq implementation of the model of equational logic, but provides only paper proofs of the validity and completeness of the logic. Additionally, while the proof theory is defined in the paper, and used in the construction of the term algebra for use in the model, a separate implementation of the deductive semantics is not provided disallowing the potential formalization of the machine verifiable proofs in Coq.

1.4.4 Coq Formalizations of Logic

Coq has also been used as a formalization mechanism for other logics including the aforementioned equational logic, Linear Temporal Logic (LTL) [16, 17], and a three-valued logic. From here, we can see other approaches to the formalizations of logic in Coq.

In [16], LTL is axiomatized via a shallow embedding in Coq where program executions are represented by infinite co-inductive lists and temporal operators are co-inductive or inductive types. This allows for expressing the temporal notions

of the logic without an explicit time parameter. Each operation in LTL is then a separate Inductive or CoInductive Prop in Coq with *trace* and *run* operations for exercising a particular theory.

Another approach is a deep embedding in Coq as seen in [17], where the syntax of LTL is defined as an inductive structure, and then the semantics of LTL is defined as satisfaction and entailment operations over the syntax. An axiomatization of LTL is then demonstrated to be sound with respect to the semantics via a series of theorems representing the axioms in Coq.

The approach used in this paper is similar to the second approach in that we create a deep embedding of rewriting logic, but instead of an axiomatic semantics, we construct a model based on universal algebra by which we validate the proof theoretic semantics.

1.4.5 Coq Formalizations of Concurrency

The interest in having an interactive theorem proving environment to express concurrent systems has also been approached with the formalization of the π-calculus in Coq.

One approach to formalizing π-calculus [18] emphasized its use for the specification of systems, and was followed with a specification of an SMTP server originally implemented in java in Coq. Another approach [19] to formalizing π-calculus in Coq, using coinductive structures, was more theoretically inclined and looked to make Coq into a generic proof editor for π-calculus.

1.5 Structure of Thesis

The remainder of this work is structured as follows.

Chapter 2 presents the basics of term rewriting, formal logic, universal algebra and the calculus of inductive constructions used and referenced throughout the rest of this work.

Section 2.1 presents a short tutorial on term rewriting which is the model of computation behind the construction of rewriting logic.

Section 2.2 contains an introduction to formal logic and the definition of logics with propositional logic as the guiding example.

Section 2.3 presents a short tutorial on universal algebra up to the definition of term and quotient algebras as used in the model theoretic semantics in Subsection 3.2.2.

Chapter 3 describes in detail the newly defined semantics for rewriting logic, both proof theoretic and model theoretic, and proves that rewriting logic is both sound and complete.

Chapter 4 presents the full Coq source for an example rewriting logic theory,and shows the use of Coq as a theorem proving environment for rewriting logic.

Chapter 5 discusses the use of the formal semantics plus the Coq theorem proving environment to construct a rewriting logic engine and tactics for automating proof in rewriting logic theories.

Chapter 6 wraps up the main body of the thesis and presents directions for future work.

CHAPTER 2

Background

We now introduce the four main concept areas used in this research which include term rewriting, formal logic, universal algebra and the calculus of inductive constructions. Term rewriting is the foundation of rewriting logic, so we briefly look at it in isolation to get a better view of our starting point. We then discuss formal logic as rewriting logic to see what rewriting logic may add to term rewriting. Universal algebra provides the mathematical objects used for a semantic model of rewriting logic. Finally, the calculus of inductive constructions, as implemented in the Coq theorem proving environment, is the vehicle with which all of this work is formalized.

2.1 Term Rewriting

Term Rewriting [20, 21, 22] is a model of computation categorized by the repeated replacement of terms from a set of directed rules. We construct a theory below in the syntax of Maude.

```
fmod PEANO-MATH is
 sorts A .
 op 0 : → A .
 op S : A → A .
 op + _ _ : A A → A .
 op * _ _ : A A → A .
 vars V1 V2 : A .
 eq + 0 V1 = V1 .
 eq + (S V1) V2 = S (+ V1 V2) .
 eq * 0 V1 = 0 .
 eq * V1 0 = 0 .
 eq * V1 (S V2) = + V1 (* V1 V2) .
endfm
```

PEANO-MATH is a formalization of peano arithmetic using a system of equations which define the addition and multiplication operators. While equations represent a bi-directional relationship, in term rewriting they are only ever applied in one direction in an attempt to produce an answer, so the right-hand side of a rule should be a reduction of some kind of the left-hand side. To trace through a rewriting of 2 + 3 below, we show the full term followed by the left-hand side of a matching rule along with the substitution enabling that match. The term resulting from applying the substitution to the right-hand side term of the rule follows. Replacement can happen for both the top level term, as seen in the first step below, as well as in inner terms, as seen in the second step.

```
  + (S S 0) (S S S 0)
      MATCH + (S V1) V2 with V1 = S 0, V2 = S S S 0
= S (+ (S 0) (S S S 0))
      MATCH + (S V1) V2 with V1 = 0, V2 = S S S 0
= S S (+ 0 (S S S 0))
      MATCH + 0 V1 with V1 = S S S 0
= S S S S S 0
```

The application of a rewrite is also known as a *reduction* as the usual intent is to produce a term that is in someway smaller than the original term. By smaller, we may a shorter length string, occurring sooner according to an alphabetical sorting, or smaller could be according to some other ordering relationship. Any well defined, well ordered relationship can give us a valid meaning of smaller in this context. The ability to produce smaller terms leads to the values of rewriting logic which are *normal forms*, or terms that cannot be reduced any further. $S\ S\ S\ S\ S\ O$ as generated in our example above is an example of a normal form.

2.1.1 Syntax and Semantics

To define a term rewriting system, we require a few syntactic components. First we need a *signature*, Σ, defining the set of allowable function symbols along with their arities. We also require a countably infinite set of variables, X, distinct from the function symbols. The set of terms, $\mathrm{T}_\Sigma(X)$, in a theory can then be inductively defined as follows:

- if x $\in$ X then x $\in \mathrm{T}_\Sigma(X)$
- if f $\in \Sigma$ is a n-ary function symbol and $s_1, ..., s_n \in \mathrm{T}_\Sigma(X)$, then f$(s_1, ..., s_n) \in \mathrm{T}_\Sigma(X)$.

The *rewrite rules* of a theory are pairs of terms, $l \to r$ with $l, r \in \mathrm{T}_\Sigma(X)$, representing equalities, and where the set of variables in the right-hand side of the rule are a subset of the set of variables in the left-hand side of the rule. We do not allow the introduction of variables in a rewrite. Additionally, l must not be a variable. In addition to terms as defined above, we also have *ground terms*, which are terms without variables. Generally, equations are composed of terms which may contain variables, but we reason, or rewrite, over ground terms.

2.1.2 Properties of Term Rewriting Theories

There are a variety of useful properties on rewriting systems that make them more useful, especially in a computation sense.

Termination: A theory is said to be terminating if for all inputs, we eventually reach an expression which doesn't match any of the rules. Such an expression

is said to be a *normal form* and is considered a minimal term in the theory. In the below term, S S S S S O as generated in our example above is in a normal form because it cannot be reduced any further.

$$\texttt{S S O} + \texttt{S S S O} \rightarrow \texttt{S S S S S O}$$

Confluence: If the order of application of rewrite operations leads to terms that are always *joinable*, then the theory is said to be confluent. A pair of terms is said to be joinable if they are syntactically equivalent, or can be made so via additional rewrites. Confluent systems therefore have unique normal forms.

(EQ1) $\texttt{a} = \texttt{b}$
(EQ2) $\texttt{a} = \texttt{c}$

Here we have a trivial example of a non-confluent rewriting system, since starting from a, we can reach b or c. To make the above system confluent, we need to add the implied equation, b = c. This is know as the *completion* of term rewriting systems. The equation is implied since a is equal to b and a is equal to c, therefore b must be equal to c, or the earlier equalities are untrue.

A system that is both terminating and confluent has a unique normal form for each term which can be produced by rewriting each term as far as possible. It is for these systems that term rewriting is a decision procedure for equational logic as equality between two terms can be determined by rewriting each term to its normal form, and then doing a straight syntactic comparison.

2.2 Formal Logic

A logic is a tool to reason about phenomena. Specifically, it is a language designed to deduce the truth of a sentence from a core set of axioms. Before discussing rewriting logic, it may be illustrative to talk about equational logic, which is closely related, to help define what we mean by a formal logic. Formally, a logic is a combination of a well defined syntax plus an accompanying semantics.

A semantics may be a set of rules for deductive proofs over sentences in the logic, also known as the *proof theory*. The proof theory is a purely syntactic construction where we can decide the truthfulness of a sentence based solely on its form. Alternatively we can build mathematical models to decide truth in a logic which is know as *model theory*.

In the rest of this section, we will formalize a portion of equational logic as a means of introducing formal logic and as the basis with which to discuss later the concepts of logic as they are applied to rewriting logic.

In equational logic, a theory is a set of equations defining relationships amongst terms. We can define the terms of equational logic as we did with term rewriting previously defining terms over a signature with variables to get the set $\mathrm{T}_\Sigma(X)$. An *equation* is then a pair of terms $l, r \in \mathrm{T}_\Sigma(X)$ which are considered equal, while an *equational theory* is a set of equations. The PEANO-MATH example given in section 2.1 can be considered an equational theory where all of the lines starting with **eq** represent the set of equations.

To determine whether or not a pair of terms is equal in a given theory, we can construct a derivation using the follow rules:

Reflexivity for each $t \in \mathrm{T}_\Sigma(X)$,

$$\frac{}{t = t}$$

Congruence For each $f \in \Sigma_n, n \in \mathbb{N}$

$$\frac{t_1 = t'_1 \quad ... \quad t_n = t'_n}{f(t_1, ..., t_n) = f(t'_1, ..., t'_n)}$$

Replacement For each equation $r : t(\overline{x}) = t'(\overline{x})$ in $\mathcal{R}$, where $\overline{x}$ is set of variables in t, and given proofs relating the set $\overline{w}$ to their respective elements in the set $\overline{w'}$ where $\overline{w}$ and $\overline{w'}$ have the same number of elements as $\overline{x}$,

$$\frac{\overline{w} = \overline{w'}}{t(\overline{w}/\overline{x}) = t'(\overline{w'}/\overline{x})}$$

$t(\overline{w}/\overline{x})$ represents binding the elements in $\overline{w}$ to the variables in $\overline{x}$.

Transitivity

$$\frac{t_1 = t_2 \quad t_2 = t_3}{t_1 = t_3}$$

Symmetry

$$\frac{t_1 = t_2}{t_2 = t_1}$$

Reflexivity states that a term is always equal to itself. Transitivity states that if the pair of terms $t1$ and $t2$ are equal, and the pair of terms $t2$ and $t3$ are equal, then $t1$ must be equal to $t3$. Equations are symmetric as might be expected. Congruence says that if the subterms of two terms are equal, then the composite terms are equal. Replacement deals with binding terms to variables and maintaining equality. These same concepts are used in defining rewriting logic in subsection 3.2.1 and are detailed further there. Taken together, these rules of *equational reasoning*

constitute a proof theory for equational logic that allow for deduction to determine equality between terms. We can envision the example of addition presented in the previous section to also be derivation in equational logic as it is also the repeated application of equations.

We can also look at universal algebra as a model for equational theories. We can define the set of terms in a theory as a *term algebra* by using the constructors of the terms as generators. We can then define a quotient term algebra by taking the rules of deduction above, along with the equations as the theory, as a partitioning operator to divide the term algebra into sets of equivalent terms. The quotient algebra generated is said to model a pair of terms $t1, t2$ in a theory E if the two terms fall in the same partition of the model.

It has been shown in [23] that equational logic is both sound and complete allowing us to to freely talk about the proof theory or model theory and have the results be relevant for the other.

Equational reasoning as formulated above is difficult to use in an automated theorem prover due to equality being a symmetric relationship. Because any equation in the theory can be used in both directions, the process of determining equality is generally non-terminating in a mechanized process. To get around this problem, theorem provers generally eliminate the symmetry rule and treat the equations as a set of directed rules. This leads to term rewriting being a mechanization of equational logic which is sound, but no longer complete. Any derivation found must still be in the model, but the model now contains equalities that our mechanization cannot find.

2.3 Universal Algebra

Universal algebra is the study of algebras, their properties and relationships [24, 25]. The notion of an algebra as a mathematical object is of particular interest to us since algebras have proven to be useful as models in formal semantics. Here we define the basics of universal algebra up to a definition of quotient term algebras which we will later use as the models of rewriting logic. A more comprehensive introduction to universal algebra can be found in

An algebra is defined to be a set together with functions of particular arities over the set. Operations of arity 0 are also called *constants*. Additionally, the functions are often constrained by a set of equational properties. For example, we can define a semigroup as an algebraic structure consisting of a set together with a binary operation which is associative. We can see from this definition that there are many concrete examples of a semigroup such as the set of natural numbers with the plus operation, or the set of Boolean values with the *logical or* operation. In these cases, the sets of natural number and booleans, respectively, are known as the *carrier sets* of the concrete algebras.

This form of algebra is known as an unsorted algebra since there is only a single carrier set. There are extensions to many-sorted, order-sorted, and more versions which allow for multiple carriers sets which may have defined relationships to each other. Operations must then be given explicit sorting information instead of requiring only arities. For the sake of this text, we will concern ourselves only with the unsorted variant of algebras.

Signatures are a means of symbolic representation of algebras. A signature, Σ, is a set of operation names with arity.

Given an algebraic signature, we can construct a *term algebra*, T_Σ, which provides a concrete model of the abstract declaration. We generate the carrier set of the algebra recursively over the operations as follows:

- for each constant symbol σ in Σ, the string $\sigma \in \mathrm{T}_\Sigma$
- for each symbol σ in Σ, where σ has arity n, and for each list $\overline{s} = s_1 \ldots s_n$ of length n, the string $\sigma(\overline{s}) \in \mathrm{T}_\Sigma$.

Given an algebraic signature, Σ, and a variable set X of pairwise, disjoint values such that forall $x \in X$, $x \notin \Sigma$, we can define the term algebra $\mathrm{T}_\Sigma(X)$ of terms with variables.

We can define various *congruence relations* on an algebra, which are equivalence relationships that are compatible with all of the operations of the algebra. A *quotient algebra* is then defined as an algebra, along with a congruence relationship that partitions the elements of the algebra into equivalence classes.

2.4 Calculus of Inductive Constructions

For this, we turn to an existent Interactive Theorem Proving Environment, Coq. Coq is a proof assistant whose language implementation, Gallina, is based on the Calculus of Inductive Constructions [26, 27]. It provides a constructive logic with higher-order type theory for formalizations, and a rich environment for constructing and automating proofs [28].

The type theory [29] underlying Coq [2, 30] includes dependent types, an extension of type theory where types can range over values. This enables the formulation of complex types such that type checking can reduce the need for theorem proving.

Below is a formalism in Coq of a polymorphic list type parametrized over its length. The head operation is only defined for lists of size greater than 0, which means that if a call to head type checks, we have statically guaranteed to never receive an empty list and proven that the error condition of calling head on an empty list never happens.

In our example Coq code, we will make use of pretty-printed symbols include → in place of "->", ⇒ in place of "=>" and ∀ in place of "forall". In our source files, the text value is used instead.

```
Module dlist.
  Inductive list : Type → nat → Type :=
  | nil : forall A, list A O
  | cons: forall A n, A → list A n → list A (S n).

  Definition head' A n (ls : list A n) :=
    match ls in (list n) return (match n with O ⇒ unit | S _ ⇒ A end) with
    | nil _ ⇒ tt
    | cons A n val rest ⇒ val
    end .

  Definition head A n (ls : list A (S n)) : A :=
     head' A (S n) ls.

  Example test_head :
     head nat 0 (cons nat 0 5 (nil nat)) = 5.
  Proof. reflexivity. Qed.
```

The above code first defines a new inductive, dependent type 'list' which has the type "Type → nat → Type". This definition is very similar to a polymorphic list definition in any modern, functional language. In the Coq syntax, the first "Type" parameter is the type this list will be parametrized over. The next parameter, nat, is then used to embed the length of the list in its type. The base constructor for the empty list, "nil : forall A, list A 0" says that for any "A" that is a "Type", "nil" constructs a 0 length list that has type "list A 0". To construct other lists,

the "cons" constructor must be used. The type of "cons", "forall A n, A → list A n → list A (S n)", can be read as follows:

For any natural number "n" and type "A", we can take an element of "A" and an list of elements in A of size "n" and construct a new list in A of size "S n".

We then define two new functions to operate on our lists, "head" and "head'". The function "head'" accepts a list of any size and returns either the first element, or the unit value in the case of an empty list to signify an error. This is almost exactly how head would be defined in a modern, functional language such as OCaml or Haskell.

"Head", on the other hand, makes use of the size embedded in the type of lists and only accepts lists with size greater than zero, as indicated by the type of its second parameter "ls : list A (S n)". What this says is that when calling "head" in a program, for it to successfully type check, it must be given a non-empty list as a parameter. Since "head'" only returns an error for empty lists, "head" will never return an error. Both of these functions require the type the list is parametrized over and the size of the list as parameters.

The auxiliary function is necessary because in Coq, the "match" operator must return a value for every possible case. In "head", we would not have a return value for the empty list case, and therefore could not use "match" to deconstruct the list.

The return specification in "head'" is used to indicate that the function has multiple possible return types, either "unit" in the empty list case, or "A" otherwise. Coq cannot infer the desire to have multiple return types from the function necessitating this extra specification.

The code is the exercised in the "Example test_head" which calls the head function on our new list type. The size and type elements are explicitly passed to head and the constructors cons and nil.

The above example is written far more explicitly than is usual in Coq for illustrative purposes. An alternative implementation would be as follows:

```
Section newlist.
  Variable A : Type.

  Inductive list : nat → Type :=
  | nil : list O
  | cons: forall n, A → list n → list (S n).

  Definition head' n (ls : list n) :=
    match ls in (list n) return (match n with O ⇒ unit | S _ ⇒ A end) with
    | nil ⇒ tt
    | cons _ val _ ⇒ val
    end .

  Definition head _ (ls : ilist (S n)) : A :=
     head' _ ls.
End newlist.

Implicit Arguments head [A n].
Implicit Arguments cons [A n].
Implicit Arguments nil [A].

Example test_head :
   head (cons 5 nil) = 5.
Proof. reflexivity. Qed.
```

First, we use a section to declare A which makes it available to use implicitly in the later declarations in the section. The type of "list" therefore changes to "nat → Type" instead of "Type → nat → Type". After defining our new type, we then make it easier to use with "Implicit Arguments" commands which instructs Coq to infer the type parameters when possible. This allows us to rewrite the sample code

as "head (cons 5 nil)" without the type information cluttering the code. In both versions of the code, "head nil" will fail to type check providing us our statically verified check against this error.

Another nice feature of Coq is its support for a small kernel proof language which allows proofs found in Coq to be independently verified by an external tool. The benefit of this is that the user doesn't need to trust the search procedures for finding a proof, because any proof found can be verified independently from the search procedure used to construct it. This allows for the development of more complex and domain specific search patterns to help automate proving in Coq without introducing the worry of bugs in the search procedures themselves.

CHAPTER 3

Rewriting Logic

Using the previously defined formalisms, we are now ready to define the syntax, proof theory and model theory of rewriting logic. Our goal is to construct and reason about sentences of the form $\mathcal{R} \vdash t \rightarrow t'$, where $\mathcal{R}$ is a rewrite theory and t, t' are a pair of terms. We read this sentence as $\mathcal{R}$ entails t becoming t' to emphasize that rewriting logic is a logic of change, or becoming.

Since rewriting logic is based on term rewriting, we are able to borrow portions of the CoLoR implementation of term rewriting for the definition of terms and same basic utilities. We then build from this base to define our proof theoretic and model theoretic semantics of rewriting logic.

3.1 Syntax

We define the syntax of *rewriting logic theories*, $\mathcal{R}$, as the pair $\mathcal{R}$= (Σ,R) where Σ is the theory signature, and R is a *rewriting relation* composed of a list of pairs terms (l, r) where $l, r \in \mathrm{T}_\Sigma(X)$, and the pairs $(l, r) \in R$ are the *rewrite rules*. A signature, Σ, is the set of allowable function symbols along with their appropriate arities.

We can define the set of terms over the signature, T_Σ, as the set of all terms constructable from combining the symbols in Σ. If we assume the existence of a countably infinite set X of variables, we can construct the set of terms with variables, $\mathrm{T}_\Sigma(X)$, as in term rewriting or equational logic.

- if x $\in$ X then x $\in \mathrm{T}_\Sigma(X)$

- if f $\in \Sigma$ is a n-ary function symbol and $s_1, ..., s_n \in \mathrm{T}_\Sigma(X)$, then f$(s_1, ..., s_n \in \mathrm{T}_\Sigma(X)$.

```
Record termSignature : Type := mkTermSignature {
  symbol : Set ;
  arity : symbol → nat ;
  beq_term_symb : symbol → symbol → bool ;
  beq_term_symb_ok : ∀ x y, beq_term_symb x y = true ↔ x = y
}.
```

To represent rewriting logic syntax in Coq, we use the **termSignature** datatype from the CoLoR term rewriting library to describe what terms look like. The Coq `Record` type is a composite type similar to a struct in *C*, using semi-colons to separate the elements and defining a helper function, mkTermSignature, for constructing **termSignature** instances. Each element is defined with a name and a type, separated by a colon. The element name is also used an accessor function to retrieve its value from a record instance.

A **termSignature** is made of up four elements starting with symbol, a Coq `Set`, which represents the set of function symbols in our signature. The arity is a function from symbol to **nat**, and represents the arities for each function symbol in the signature. Then beq_term_symb holds a boolean equality function for comparing symbols while beq_term_symb_ok is a proof that beq_term_symb returns true if and only if the symbols are considered equal in Coq, which means they are syntactically equal. The later two functions are included in the term signature for convenience and not necessity, as external function definitions would be suitable.

Instead of using strings to represent the function symbols, as is more typical for term rewriting, we make use of the Coq type checker with the definition of symbol

as a `Set`. It is expected that the type used for symbol contains a number of 0-arity constructors representing the allowable terms in Σ. This can be seen in the definition of *symbols* in Chapter 4 where it is used to construct the *termSig* term signature as part of the rewriting logic theory of peano numbers.

This formalization allows for easier proofs over the signatures in later developments by having the Coq type system enforce that only symbols from the allowable set are being used to construct terms. Using arbitrary strings would remove the compile time type checking and introduce an avenue for errors.

`Implicit Arguments` mkTermSignature [*symbol beq_term_symb*] .
Arguments arity [*t*] _ .

As a further nod to convenience, we can declare certain arguments to be implicit parameters such as the type of symbol in the arity function. This type information will be inferable from the other arguments to arity and therefore we make it so that it does not need to be explicitly mentioned. In the above, we've also declared that symbol and beq_term_symb do not need to be manually specified when constructing a term signature. The type of symbol can be inferred from the type signature for the arity function, and the type of beq_term_symb can be inferred from the type of beq_term_symb_ok. We will continue to create implicit arguments as appropriate throughout the code, but will only mention it if the usage is at all ambiguous or unclear.

`Notation` variable := **nat** (*only parsing*).

Before we define terms, we must first define the notion of a variable so that we can define the terms in $\mathrm{T}_\Sigma(X)$. For our purposes, the set of nats meets all of our variable requirements, being a countably infinite set of distinct members. To

keep variables visually distinct, while still retaining all of the benefits of using nats including existing proofs, we define our *variable* type as a notational representation for nats. Internally to Coq, variable is interchangeable with **nat**, so for our development we can use variable as our type.

```
Variable Sig : termSignature .
Inductive term : Type :=
| Var : variable → term
| Fun : ∀ f : (symbol Sig), vector term (arity f) → term.
Notation terms := (vector term) .
```

We define **term** to be an inductive data type with two constructors, Var and Fun. Coq uses a vertical bar to separate constructors in an inductive data type, and here we preface the constructors with this syntax as a stylistic choice. Terms are parameterized over a **termSignature**, which we accomplish above with the `Variable` *Sig* Coq command, so all of the constructors defined for this type also implicitly take a **termSignature** as their first argument.

A rewriting logic term is either a variable, for which we define the constructor Var which takes a variable as a parameter, or a term is a function symbol. To define function symbols, we define the constructor Fun which takes two additional arguments, *f*, which is any element in set of symbols in our signature *Sig*, and a vector of terms whose length is defined by the arity of the symbol *f* in the signature *Sig*.

```
Record rewrite_rule : Type := mkRule { lhs : term; rhs : term } .
Definition rwl_theory := list rewrite_rule .
```

We can now directly represent rewrite rules as a pair of terms in a signature. Again

making use of the Coq record type, a **rewrite_rule** is a pair of terms we label lhs and rhs and whose accessor functions are *lhs* and *rhs* respectively.

Finally, we can define the two-tuple $(\Sigma, \mathcal{R})$, the formal signature of a rewriting logic theory. Since the term signature is still being added to all of our definitions, a rewriting logic theory, *rwl_theory*, is a list of rewrite rules, implicitly over the termSignature Σ.

3.2 Semantics

3.2.1 Proof Theory

The rules of deduction for rewriting logic are a relationship between a rewriting theory $\mathcal{R}$, and a pair of terms $t, t' \in \mathrm{T}_\Sigma(X)$, showing how one term can become another, expressed as $\mathcal{R} \vdash t \to t'$. We say that t can become t' if and only if a proof can be obtained via finite application of the rules of deduction below:

Reflexivity for each $t \in \mathrm{T}_\Sigma(X)$,

$$\frac{}{t \to t}$$

Congruence For each $f \in \Sigma_n, n \in \mathbb{N}$

$$\frac{t_1 \to t'_1 \quad ... \quad t_n \to t'_n}{f(t_1, ..., t_n) \to f(t'_1, ..., t'_n)}$$

Replacement For each rewrite rule $r : t(\overline{x}) \to t'(\overline{x})$ in $\mathcal{R}$, where $\overline{x}$ is set of variables in t, and given proofs relating the set $\overline{w}$ to their respective elements in the set $\overline{w'}$ where $\overline{w}$ and $\overline{w'}$ have the same number of elements as $\overline{x}$,

$$\frac{\overline{w} \to \overline{w'}}{t(\overline{w}/\overline{x}) \to t'(\overline{w'}/\overline{x})}$$

$t(\overline{w}/\overline{x})$ represents binding the elements in $\overline{w}$ to the variables in $\overline{x}$.

Transitivity

$$\frac{t_1 \to t_2 \quad t_2 \to t_3}{t_1 \to t_3}$$

Reflexivity states any term in $\mathrm{T}_\Sigma(X)$ can be be derived from itself. *Congruence* provides for rewriting below the top level term by stating that any term constructed from a function symbol with the appropriate number of subterms can become a new term with the same function symbol where each of the subterms is potentially rewritten. This also provides our first ability to model concurrency since we can simultaneously rewrite all of the subterms involved in the action. *Transitivity* is the composition of rewrites which states if we can rewrite some term t_1 into another term t_2, and additionally if we can rewrite t_2 to a term t_3, then we can rewrite t_1 to t_3.

Actual replacement, along with the handling of variable is dealt with in the *replacement* rule. Here, $\overline{x}$ represents the set of variables in the term $t(\overline{x})$, and $\overline{w}$ represents a collection of terms in $\mathrm{T}_\Sigma(X)$. $t(\overline{w}/\overline{x})$ then represents the term $t(\overline{x})$ with the variables bound to the terms in $\overline{w}$. We then say if the terms in $\overline{w}$ can be rewritten to the set of terms $\overline{w}'$, and the term $t(\overline{w}/\overline{x})$ matches the left-hand side of a rule $r \in \mathcal{R}$, then we derive the term $t'(\overline{w}'/\overline{x})$ which is the right-hand side of r with its variables replaced by the corresponding terms in $\overline{w}'$. Replacement provides an alternative form of concurrency since we can simultaneously rewrite a term, and all of the terms in the current substitution.

We transpose these rules into an inductive datatype in Coq parametrized by the termSignature Σ as Sig, a rewrite theory and a pair of terms. Each potential rule of deduction in rewriting logic is then a constructor of this property with the necessary recursive properties over lists and substitutions embedded. The constructors create proof objects in Coq, as evidenced by the type living in *Prop*.

```
Inductive proof_theory : rwl_theory Sig → term Sig → term Sig → Prop :=
  | PF_Refl : ∀ t1 RWT, RWT ⊢ t1 → t1
  | PF_Congruence : ∀ RWT f tl tl2,
                        proof_theory_list (arity Sig f) RWT tl tl2 →
                        (RWT ⊢ Fun f tl → Fun f tl2)


  | PF_Replacement : ∀ RWT inputTerm outputTerm rule Subs Subs',
                         ruleIn rule RWT →
                         eq (applySubstitution Subs (lhs rule)) inputTerm →
                         eq (applySubstitution Subs' (rhs rule)) outputTerm →
                         proof_theory_substitutions RWT Subs Subs' →
                         RWT ⊢ inputTerm → outputTerm
  | PF_Transitive : ∀ t1 t2 t3 RWT,
                        (RWT ⊢ t1 → t2) → (RWT ⊢ t2 → t3) → (RWT ⊢ t1 →
t3)
```

The first transition rule is *PF_Refl*, representing reflexivity, and states that any term in the rewrite theory is reachable from itself.

Next we have the congruence rule, *PF_Congruence*, which states that if we have a term comprised of a function symbol f, and its set of subterms tl, we can simultaneously reduce each of the subterms to create a new subterm list $tl2$ and the resultant term $f(tl2)$ is reachable from $f(tl)$ in the rewrite theory. This is accomplished via the proof_theory_list rules which maps the proof theory over all the terms in a vector of terms.

The third rule of deduction in rewriting logic is *PF_Replacement*, representing replacement, which is the only rule to directly work with the rewrite rules in the theory. The replacement rule requires us to find a rule in our rewrite theory such that the current term of interest matches the left-hand side of the rule with an appropriate substitution. We can then prove the derivation of the right hand side of the rule in the same substitution. Additionally, the rules in proof_theory_substitution states the the substitution need not be identical, but rather we use a new substitution where the corresponding terms are derivable from their counterparts in the original substitution. This allows for concurrent rewrites by simultaneously rewriting a term and its substitution.

To complete the reducibility relationship, we need to introduce *PF_Transitivity*, the transitivity rule which allows for multi-step derivations. For any two terms $t, t' \in \mathcal{R}$, if there is another term $t'' \in \mathcal{R}$ such that $\mathcal{R} \vdash t \to t''$ and $\mathcal{R} \vdash t'' \to t'$, then we can say that $\mathcal{R} \vdash t \to t'$. If $\mathcal{R} \vdash t \to t'$ without use of the transitive relationship, then we can say there is a one-step concurrent rewrite from t to t'. If there is only one use of the replacement rule, then we can say that there is a sequential rewrite from t to t'.

These four rules map directly from our previous formalization of the proof theory. However, to make this a valid data type in Coq, we must also formalize the notions of **proof_theory_list** and **proof_theory_substitutions** which we took for granted earlier. Since elements in **proof_theory** contain these types, and these types contain elements of **proof_theory**, we create a mutually inductive datatype by using the `with` keyword to continue out definition.

`with` **proof_theory_list** : $\forall\ n :$ **nat**, rwl_theory *Sig* $\to$ vector (**term** *Sig*) $n \to$ vector (**term** *Sig*) $n \to$ `Prop` :=

```
  | PF_Nil : ∀ RWT, proof_theory_list 0 RWT Vnil Vnil
  | PF_Cons : ∀ RWT, ∀ m, ∀ (rest1 rest2 : vector (term Sig) m), ∀ t1 t2,
                proof_theory RWT t1 t2 →
                proof_theory_list m RWT rest1 rest2 →
                proof_theory_list (S m) RWT (Vcons t1 rest1) (Vcons t2 rest2)
with proof_theory_substitutions : rwl_theory Sig → substitution Sig → substitution Sig → Prop :=


  | PF_ReflSub : ∀ RWT st, proof_theory_substitutions RWT st st
  | PF_HeavyLeft : ∀ RWT entry sub sub',
                              proof_theory_substitutions RWT sub sub' →
                              proof_theory_substitutions RWT (entry :: sub) sub'
  | PF_ConsSubstitution : ∀ RWT id1 t1 t2 rest1 rest2,
                                proof_theory RWT t1 t2 →
                                proof_theory_substitutions RWT rest1 rest2 →
                                proof_theory_substitutions RWT ((id1, t1) ::
rest1) ((id1, t2) :: rest2)
  where "RWT '—-' t1 '-¿' t2" := (proof_theory RWT t1 t2).
```

We formalize subterm lists as a Coq vector that limits the length based on the arity of the function symbol. We say that our theory holds for a pair of lists if either the lists are empty, or if the head of both lists represents a proveable relationship, and the rest of the lists recursively hold. Similarly, we say that the theory holds for two substitutions if the substitutions are both empty, or if the proof theory holds for the term at the head of each substitution for matching variables, and the rest of the substitutions recurively hold. For substitutions we also allow the left-hand side substitution to contain more elements than the right-hand side which we saw an example of in the definition of the *choose* operator earlier.

We now have a suitable representation of our deductive semantics for rewriting logic in Coq to enable the use of Coq as a theorem proving environment as we demonstrate in Chapter 4. There we work through constructing a rewriting logic

theory in our defined syntax, and then using the proof theory above, we show the manual derivation of some terms.

3.2.2 Model Theory

Our proof theory defines how to derive new terms with our rewriting logic theories, but it does not address the question of truthfulness. For that, we need to define a model such that we can evaluate when a derivation is true. We want to define a model such that we can say that for a rewriting logic theory $\mathcal{R}= (\Sigma, Rules)$, and terms $t, t' \in \Sigma$, that we have a model $M_{\mathcal{R}}$ which models the relationship $t \rightarrow t'$, or $M_{\mathcal{R}} \models t \rightarrow t'$. Given the fact that rewriting logic is based on term rewriting, and there are existing models for equational logic in term rewriting, we start there for our model search.

From equational logic, we have as a model a quotient term algebra derived from the equational theory specification. In brief, for a given theory, we can define a term algebra where for each operation, there is a corresponding generator function, and then have as a carrier set the set of all strings generated by the functions. We can then define a quotienting relation over the set of terms by converting from the derivation rules in the proof theory for equational logic presented in section 2.2 and the set of equations in the theory.

Since equality is a symmetric relation, this of course is not a valid model for rewriting logic. For a theory $\mathcal{R}$, and terms $A, B \in \mathcal{R}$, we don't want to have $\mathcal{R} \models A \rightarrow B$ implies $\mathcal{R} \models B \rightarrow A$, which is what the above quotienting relation provides. However, it would be nice to have a similar style model.

The key in deriving a new quotient algebra is the realization that we are not interested in comparing terms. What we really want to quotient over, are the

relationships $M_{\mathcal{R}} \models t \to t'$. Our quotienting relationship then answers the question, not when are two terms equal, but rather when are two concurrent rewrites equivalent. Instead of having terms as the elements of our carrier set, what we really want are proof terms of the form $M_{\mathcal{R}} \models t \to t'$. Rewriting with the rules in a rewriting logic theory doesn't define the quotienting relationship, but rather defines the generation functions to create our carrier term algebra.

By viewing the derivation rules of rewriting logic as generating rules to generate a term algebra of proof objects, we can define a term algebra of proof terms to act as the carrier set for a new quotient algebra model of rewriting logic.

Identities for each $t \in \mathrm{T}_\Sigma(X)$,

$$\frac{}{t : t \to t}$$

Σ-structure For each $f \in \Sigma_n, n \in \mathbb{N}$

$$\frac{\alpha_1 : t_1 \to t'_1 \quad ... \quad \alpha_n : t_n \to t'_n}{f(\alpha_1, ..., \alpha_n) : f(t_1, ..., t_n) \to f(t'_1, ..., t'_n)}$$

Replacement For each rewrite rule $r : t(\overline{x}) \to t'(\overline{x})$ in $\mathcal{R}$, where $\overline{x}$ is set of variables in t, and for $\overline{\alpha}$, a set of proof objects relating the set $\overline{w}$ to their respective elements in the set $\overline{w'}$ with the same number of elements as $\overline{x}$,

$$\frac{\overline{\alpha} : \overline{w} \to \overline{w'}}{r(\overline{\alpha}) : t(\overline{w}/\overline{x}) \to t'(\overline{w'}/\overline{x})}$$

where $t(\overline{w}/\overline{x})$ represents binding the elements in $\overline{w}$ to the variables in $\overline{x}$.

Composition

$$\frac{\alpha : t_1 \to t_2 \quad \beta : t_2 \to t_3}{\alpha; \beta : t_1 \to t_3}$$

These rules are directly converted from the proof theory where we derivation rule now instead generates a proof object, and this set of proof objects forms the term algebra for a given rewriting logic theory. We can then define a number of quotienting relations providing distinct interpretations of the equivalence of a concurrent rewrite.

On the most strict end, we can define as our quotienting relation syntactic equality. Two rewrites are considered identical only when they have the exact same form. On the most permissive side, we can define as our quotienting relation any two proofs terms are considered equal if they map the same input to the same output. In other words, we have behavioral equivalence where we consider the transition to be a black box and equality is based solely on the start and end points.

In between, we can define a variety of other notions of equality, or partial notions to be combined. For instance, we might decide that transitive is an associative relationship and have equality based on that. To formalize these models in Coq, we must first convert our proof theory to set a of generator functions to generate the new term algebra. This can be accomplished most easily by creating a new models inductive datatype living in the Coq type "Type" instead of "Prop" demonstrating that it represents objects instead of proofs. Those objects are the generated proof terms in rewriting logic.

```
Inductive models : rwl_theory Sig → term Sig → term Sig → Type :=
  | M_Refl : ∀ t1 RWT, models RWT t1 t1
  | M_Congruence : ∀ RWT f tl tl2,
                        lmodels (arity Sig f) RWT tl tl2 →
                        models RWT (Fun f tl) (Fun f tl2)


  | M_Replacement : ∀ RWT inputTerm outputTerm rule Subs Subs',
                          ruleIn rule RWT →
```

```
                    eq (applySubstitution Subs (lhs rule)) inputTerm →
                    eq (applySubstitution Subs' (rhs rule)) outputTerm →
                    subsModel RWT Subs Subs' →
                    models RWT inputTerm outputTerm
  | M_Transitive : ∀ t1 t2 t3 RWT,
                   (models RWT t1 t2) →
                   (models RWT t2 t3) →
                   (models RWT t1 t3)
```

We implement our term algebra generator functions in Coq as an inductive datatype in `Type` to represent that these are objects representing derivations instead of being Coq proofs. The four rules of generation map to the four constructors of the **models** type.

```
with lmodels : ∀ n : nat, rwl_theory Sig → vector (term Sig) n → vector (term
Sig) n → Type :=
  | M_Nil : ∀ RWT, lmodels 0 RWT Vnil Vnil
  | M_Cons : ∀ RWT, ∀ m, ∀ (rest1 rest2 : vector (term Sig) m), ∀ t1 t2,
             models RWT t1 t2 →
             lmodels m RWT rest1 rest2 →
             lmodels (S m) RWT (Vcons t1 rest1) (Vcons t2 rest2)
with subsModel : rwl_theory Sig → substitution Sig → substitution Sig → Type
:=
  | M_ReflSub : ∀ RWT st, subsModel RWT st st
  | M_HeavyLeft : ∀ RWT entry sub sub',
                  subsModel RWT sub sub' →
                  subsModel RWT (entry :: sub) sub'
  | M_ConsSubstitution : ∀ RWT id1 t1 t2 rest1 rest2,
                    models RWT t1 t2 →
                    subsModel RWT rest1 rest2 →
                    subsModel RWT ((id1 , t1) :: rest1) ((id1 , t2)
:: rest2).
```

The term algebra requires substitutions and lists of terms just as the proof theory did. We again implement these as a mutally inductive type in Coq where **lmodels** represents list of terms and **subsModel** represents substitutions in our term

algebra.

The elements generated from the types above form the term algebra carrier set for our quotient algebra model of rewriting logic. We begin constructing the quotient algebra by defining the algebraic signature. As we are defining a term algebra for the unsorted rewriting logic, our algebraic signature requires only a single sort and no methods.

```
Definition algSig : Signature := single_sorted_signature (False_rect _) .
```

As parameters to our model, we require a rewriting logic theory signature, which is decomposed into the term signature and set of rewrite rules. Since these components can be deferred until we are constructing a model for a particular rewrite theory, we specify them as variables in our model and use them as *ts* for the term signature and *rwt* to represent an arbitrary theory in the logic over the signature.

The **models** generator above is defined to take the two input terms as separate parameters. We need them to be a singular parameter for the subset construction below, so we define a wrapper type of pairs of terms which generates exactly the set of term pairs that models generates. It simply states that for all theories if **models** generates a pair of terms with associated proof object, then **generates** does as well.

```
Inductive generates {s : termSignature } (r : rwl_theory s) : (prod (term s)
(term s)) → Type :=
  | can_gen : ∀ t t' : term s, models s r t t' → generates r (t, t') .
```

We can now define the carrier set for our term algebra as the set of pairs of terms

plus associated proof object generated by our generator functions, the constructors of the model type. The type below should be read as the set of all pairs of terms over the signature *ts* such that **generates** return that pair plus a proof object of that pair in the given theory *rwt*.

`model_carrier` = {`X` : `term ts` ∗ `term ts` & `generates rwt X`}

Finally, we can define the quotient term algebra as the single-sorted algebra whose elements are a triple containing t, t', and pf_t_t' where t and t' are two terms and pf_t_t' is the generated proof object between these two terms which has been generated from the rules of rewriting logic ported to generator functions, or constructors in Coq. We have as our default quotienting function **eq**, or syntactic equality below. To represent other models in our model theory, we can subsitute alternative quotienting relationships for **eq** when defining model_quotienting. We then combine the pieces into a full Coq representation of the algebra with rwlQuotientAlgebra.

```
Instance model_quotienting : Equiv model_type := eq .
Instance rwlQuotientAlgebra : Algebra algSig model_carrier .
```

Now that we've defined a generic model for rewriting logic, we can say what it means for a sequent to be satisifed by a rewrite theory. In particular, the sequents (sentences) of rewriting logic are pairs of terms, $t1, t2 \in \mathrm{T}_\Sigma(X)$, indicating that a concurrent rewriting relationship from $t1$ to $t2$ exists. For a theory $\mathcal{R}$, a model $M_\mathcal{R}$ containing the pair $t1, t2$ along with a proof object representing the relationship between $t1$ and $t2$ models the concurrent rewrite t1 $\rightarrow$ t2.

$$M_\mathcal{R} \models t1 \rightarrow t2$$

Sentences of this form are said to be satisifed in the model if the triple $(t1, t2, proof)$ exists in the carrier set. ie, a sentence is satisified if we can generate a proof object rewriting t1 to t2. To represent this in Coq, we need to define three relationships. One for each of **models**, **lmodels** and **subsModel**, since these three types are intertwined. We can read the definition of mSat as for a given theory r, a pair of terms $t, t' \in \mathrm{T}_\Sigma(X)$ is satisfied by all models of the theory if there exists an object in the term algebra generated by **models**. We then read lmSat and subSat similarly for lists of terms and substitutions respectively.

```
Section Satisfaction .
  Variable s : termSignature .
  Variable r : rwl_theory s .

  Definition mSat (t t' : term s) : Prop :=
    ∃ x : models s r t t', True .

  Definition lmSat {n} (lt lt' : vector (term s) n) : Prop :=
    ∃ x : lmodels s _ r lt lt', True .

  Definition subSat ( sub sub' : substitution s) : Prop :=
    ∃ x : subsModel s r sub sub', True .
End Satisfaction .
```

Currently we've shown a notion of syntactic equality between terms, and proof objects between terms, and therefore all elements in the model are not considered equal unless they contain syntactically equivalent proof objects. This is a very strict notion for comparing concurrent rewrites. To get at a deeper meaning of when two concurrent rewites are equal, we can look at other model instances in our family of quotient term algebra models by changing the model_quotienting relationship. This does not change the generator functions, so anything satisfiable in one model, will be satisfied by all models.

3.3 Soundness and Completeness

3.3.1 Soundness

Given our notion of satisfaction, we can now define soundness for rewriting logic.

Theorem 3.3.1. *For a rewrite theory $\mathcal{R}$, and all models $M(\mathcal{R})$,*

$$\mathcal{R} \vdash [t(\bar{x})] \rightarrow [t'(\bar{x'})]$$

implies

$$M(\mathcal{R}) \models [t(\bar{x})] \rightarrow [t'(\bar{x'})]$$

.

Or in the langauge of Coq.

```
Theorem rwl_sound : ∀ (Sig : termSignature)
                      (rwt : rwl_theory Sig)
                      (t t' : term Sig),
    proof_theory Sig rwt t t' → mSat Sig rwt t t' .
```

For any given quotient algebra $\mathcal{Q}$, we have to show that for each proof term

$$p : [t(\bar{x})] \rightarrow [t'(\bar{x})]$$

there exists a corresponding proof object in our algebra. In this case, we're saying that our generator can produce such a proof object, as all models have the same collection of proof objects, and the only difference is equality amongst the objects.

We can proceed by induction over the structure of the proof term, and show how a corresponding proof object can be generated in the model.

When creating this proof in Coq, we have to generate mutual induction principles so that proofs over proof_theory are also aware of proof_theory_list and proof_theory_substitutions as these are all distinct objects in Coq. By default, the generated induction principles are unaware that these are mutually inductive, and thus generate proof obligations that cannot be satisified. As this is a common need, Coq provides the `Scheme` command which can generate proper mutual induction principles when given a list of the relevant inductive types.

```
Scheme proof_theory_mut := Minimality for proof_theory Sort Prop
with proof_theory_list_mut := Minimality for proof_theory_list Sort Prop
with proof_theory_substitutions_mut :=
  Minimality for proof_theory_substitutions Sort Prop .
```

With our new mutually inductive propositions in hand, we can begin our proof over the structure of the proofs in the proof theory and show how for every proof, there is a corresponding object generated by the model. Given that the structure of the proof theory and the structure of the model generator are identical, our proof comes done to showing that for each proof, there is an equivalent model generator to create such proof objects. We will show the entire proof script below, but only annotate a few cases for illustrative purposes.

```
Proof .
intro Sig .
  apply (proof_theory_mut Sig
   (fun rwt ⇒ fun t ⇒ fun t' ⇒
      mSat Sig rwt t t')
   (fun n ⇒ fun rwt ⇒ fun t ⇒ fun t' ⇒
      lmSat Sig rwt t t')
   (fun rwt ⇒ fun sub ⇒ fun sub' ⇒
```

```
    subSat Sig rwt sub sub')
  ) ; unfold mSat .
```

The proof begins by application of induction over the structure of our proof theory. Since our proof theory is mutually recursive with the vector and substitution version, we must apply our previously defined mutual induction principles and manually specify the property we are looking for in each of the three cases. We use the three satifiability properties defined along with our model theory to specify we need model objects, lists of model objects, and substitutions containing model objects.

Then we proceed to the first case which is reflexivity, where every proof object can generate itself. The corresponding goal for reflexivity is:

```
forall (t1 : term Sig) (RWT : rwl_theory Sig),
exists _ : models Sig RWT t1 t1, True
```

which says that for all rewriting logic theories *RWT* over a signature *Sig*, we should have in our model an object for all syntactically valid terms, *t1*, to themselves. This can be shown with the existence of the M_Refl proof object generated by the model. To construct this proof in Coq, we first introduce the theory *RWT* and the term *t1* as variables, then post the existance of M_Refl with *RWT* and *t1* as inputs. This is the general form all of the soundness proofs will take.

Case "Reflexivity" .
`intros` *t1 RWT* . ∃ (M_Refl *Sig t1 RWT*) . `tauto` .

Next we show the proof obligation for the congruence case.

```
forall (RWT : rwl_theory Sig) (f : symbol Sig)
  (tl tl2 : vector (term Sig) (arity Sig f)),
proof_theory_list Sig (arity Sig f) RWT tl tl2 →
lmSat Sig RWT tl tl2 →
```

```
exists _ : models Sig RWT (Fun f tl) (Fun f tl2), True
```

As usual, our proof is over all theories *RWT*, but instead of the singular term in the reflexivity case, we must now show that for all function symbols *f*, and pairs of vectors of terms *tl* and *tl2* who have length equal to the arity of *f*, that if we can provide pairwise proofs from the term in *tl* to the terms in *tl2*, and have a model object corresponding to those proofs, then we have a model object relating *f tl* and *f tl2*.

```
Case "Congruence" .
intros RWT f tl tl2 H0 H1 .
  unfold ImSat in H1 .
  destruct H1 .
  ∃ (M_Congruence Sig RWT f tl tl2 x) .
  auto .
```

The congruence proof derivation requires the use of proof_theory_list in its inductive step to show that each subterm of the top level term has a corresponding proof object. Under the assumption that soundness holds for each subterm, we show that the model can generate a congruence proof object for any top level term created with these subterms.

We assume, for now, that soundness holds for lists of models and introduce as a hypothesis *H1*, the model object relating *tl* and *tl2* which ends up in our hypothesis as *x*. We can now show that congruence is sound by positing the M_Congruence object with *RWT*, *f*, *tl*, *tl2* and the model object *x*. This assumption is a proof obligation that we will discharge later.

The replacement case proceeds similarly to the congruence case, but relies on proof_theory_substitutions for its inductive steps.

```
Case "Replacement".
intros RWT inputTerm outputTerm rule Subs Subs' rIn lhsEq rhsEq subRed subSatObj .
  unfold subSat in subSatObj .
  destruct subSatObj .
  ∃ (M_Replacement Sig RWT inputTerm outputTerm rule Subs Subs' rIn lhsEq rhsEq x) .
  auto .
```

For replacement, we assume some theory *RWT*, a pair of terms *inputTerm* and *outputTerm*, a rule *rule*, and a pair of substitutions *Subs* and *Subs'*. We also assume a proof that *rule* is in *RWT*, and proofs that *inputTerm* and *outputTerm* can be constructed from the rule with the substitutions *Subs* and *Subs'* respectively. Finally, we assume that the model is sound in substitutions, though that generates a proof obligation to the solved later.

The last case in basic model is transitivity. Here we have to show that given a model object relating two terms *t1* and *t2*, and another model object relating *t2* to a third term *t3*, there exists a model object relating *t1* to *t3*. We can construct such an object via the M_Transitive generator completing the soundness proof for the basic proof terms.

```
Case "Transitivity" .
intros t1 t2 t3 RWT r12 e12 r23 e23 .
  destruct e12 . destruct e23 .
  ∃ (M_Transitive Sig t1 t2 t3 RWT x x0) .
  auto .
```

After proving all of the primary cases from the proof theory, we must also show that the relationship holds when dealing with our mutually inductive list of proof objects, which amounts to the same style of proof as above, and makes use of the fact the the proof theory is also being proven sound.

There are two cases, the *nil*, or empty list, and the *cons* list which contains at least one element. We can construct proof objects with the M_Nil and M_Cons generators respectively with the assumption that in the *cons* case, the pairwise elements have proof objects. We have proven this to be true above by showing soundness in the single term cases.

```
Case "List Nil" .
intros . ∃ (M_Nil Sig RWT) . auto .
Case "List Cons" .
intros .
  destruct H0 .
  unfold ImSat in H2 ; destruct H2 .
  ∃ (M_Cons Sig RWT m rest1 rest2 t1 t2 x x0) ; auto .
```

Finally, we need to repeat the exercise for our substitutions. These proofs would generally be less explicit when done in a purely textual setting, but this level of detail is required for the computer verification.

```
Case "Substitution Reflexive" .
intros . unfold subSat . ∃ (M_ReflSub Sig RWT st) . auto .
Case "Substitution HeavyLeft" .
intros . unfold subSat in *. destruct H0 .
  ∃ (M_HeavyLeft Sig RWT entry sub sub' x) . auto .
Case "Substutition Cons" .
intros . unfold subSat .
  destruct H0 .
  unfold subSat in H2 . destruct H2 .
  ∃ (M_ConsSubstitution Sig RWT id1 t1 t2 rest1 rest2 x x0 ).
  auto .
Qed.
```

3.3.2 Completeness

Theorem 3.3.2. *(Completeness). For a rewrite theory $\mathcal{R}$, and all models $M(\mathcal{R})$,*

$$M(\mathcal{R}) \models [t(\bar{x})] \rightarrow [t'(\bar{x'})]$$

implies

$$\mathcal{R} \vdash [t(\bar{x})] \rightarrow [t'(\bar{x'})]$$

The proof of completeness mirrors the proof of soundness in Coq. Since

$$M(\mathcal{R}) \models [t(\bar{x})] \rightarrow [t'(\bar{x'})]$$

We have that there is a proof object in our term algebra representing exactly the derivation we are seeking in our proof theory. To demonstrate this in Coq, we proceed by induction over the structure of the model generated proof objects. We again require a new mutual induction principle much like with the proof theory above.

```
Scheme models_mut := Minimality for models Sort Prop
with lmodels_mut := Minimality for lmodels Sort Prop
with subsModel_mut := Minimality for subsModel Sort Prop .

Theorem rwl_complete :
  ∀ (Sig : termSignature) (rwt : rwl_theory Sig) (t t' : term Sig),
    mSat Sig rwt t t' →
    proof_theory Sig rwt t t' .
intro Sig .
unfold mSat . destruct 1 .
apply (models_mut Sig
   (fun rwt ⇒ fun t ⇒ fun t' ⇒
      proof_theory Sig rwt t t')
   (fun n ⇒ fun rwt ⇒ fun t ⇒ fun t' ⇒
      proof_theory_list Sig _ rwt t t')
   (fun rwt ⇒ fun sub ⇒ fun sub' ⇒
      proof_theory_substitutions Sig rwt sub sub')
   ) .
```

The basic cases in our model generator mirror those in our proof theory, and we discharge the proof application in a similar fashion. Now, instead of showing the

existence of objects in our model, we instead show that for each object in our model, we can create a proof derivation in our proof theory. We can do this by directly applying the proof theory constructors to our generated proof obligations. Every proof case is therefore relatively straightforward.

Case "Identity" .
`intros` *t1 RWT* . `apply` PF_Refl .

The Σ-structure and Replacement cases again require using our mutual induction principles to operate over lists and substitutions respectively.

Case "Sigma-structure" .
`intros` . `apply` PF_Congruence ; `assumption` .
Case "Replacement" .
`intros` *RWT inputTerm outputTerm rule Subs Subs' rIn lhsEq rhsEq subsMod subsReach* .
  `apply` (PF_Replacement _ _ _ _ *rule Subs Subs'*) ; `assumption` .
Case "Transitivity" .
`intros` *t1 t2 t3 RWT H1 H2 H3* . `apply` PF_Transitive ; `assumption` .

We must then create explicit proofs for our lists of proof objects and substitions to satisfy the mechanical verification. None of these cases has a particularly interesting proof required.

Case "List Nil" .
`intros` . `apply` PF_Nil .
Case "List Cons" .
`intros` . `apply` PF_Cons ; `assumption` .
Case "Substitution Reflexive" .
`intros` . `apply` PF_ReflSub ; `assumption` .
Case "Substitution HeavyLeft" .
`intros` . `apply` PF_HeavyLeft ; `assumption` .
Case "Substitution Cons" .
`intros` . `apply` PF_ConsSubstitution ; `assumption` .
`assumption` .
`Qed`.

3.3.3 Sound and Complete

Finally we can construct a proof object showing that with our model, rewriting logic is sound and complete.

$$\mathcal{R} \vdash [t(\bar{x})] \rightarrow [t'(\bar{x})] \quad \Leftrightarrow \quad M(\mathcal{R}) \models [t(\bar{x})] \rightarrow [t'(\bar{x})]$$

This proof simply applies the two proofs constructed above.

```
Theorem rwl_sound_and_complete :
  ∀ (Sig : termSignature) (rwt : rwl_theory Sig) (t t' : term Sig),
    proof_theory Sig rwt t t' ↔
    mSat Sig rwt t t' .
split .
apply rwl_sound .
apply rwl_complete .
Qed .
```

CHAPTER 4

Example

Our embedded proof theory allows us to now use Coq as a generic theorem proving environment for rewriting logic. We begin by defining a simple rewriting logic theory that allows us to demonstrate that rewriting logic can model non-determinism. The theory is peano numbers along with a non-determinism choose operation that can become either of its inputs. For comparison, the equivalent Maude code would be:

```
mod natPlus is
  sorts S .
  op 0 : → S .
  op S : S → S .
  op choose _ _ : S S → S .
  vars V1 V2 : S .
  rl choose V1 V2 ⇒ V1 .
  rl choose V1 V2 ⇒ V2 .
endm

rewrite in natPlus : choose 0 S(0) .
> result S: 0
```

We define a single sort S to mimic the unsorted nature of our implementation. We then define two operations, O and S, which represent zero and successor and are arity 0 and 1 repectively. We also define a third operation, *choose*, of arity 2. Using these function symbols, we define the rules of this theory which state that we can choose either of the two operands when rewriting.

To implement a similar theory in Coq, we first define the set of symbols in our theory, along with a function mapping each symbol to its desired arity.

```
Inductive symbols : Set :=
| plus : symbols
| choose : symbols
| succ : symbols
| zero : symbols .
Fixpoint arity (s : symbols) : nat :=
match s with
| plus ⇒ 2
| choose ⇒ 2
| succ ⇒ 1
| zero ⇒ 0
end .
```

To complete the theory signature, we also need an equivalence operation which can distinguish the symbols, and a proof that this operation matches the Coq notion of Leibnitz equality where elements are the same only when the are syntactically equivalent.

```
Fixpoint beq_symb (s1 s2 : symbols) : bool :=
match s1, s2 with
| plus, plus ⇒ true
| choose, choose ⇒ true
| succ, succ ⇒ true
| zero, zero ⇒ true
| _, _ ⇒ false
end .
Lemma beq_symb_ok : ∀ f g : symbols, beq_symb f g = true ↔ f = g .
Proof.
Qed .
```

With the pieces in place, we can now create our term signature termSig, and then start constructing the rewriting rules to construct the theory signature. We can define utility functions tS and tchoose for generating terms below, as well variables such as v1 and v2. This is by no means as clean or straight forward as working in a language like Maude, but no effort has yet gone into syntactic sugar.

```
Definition termSig := mkTermSignature arity beq_symb beq_symb_ok .
Definition T := term termSig .
Definition tO := @Fun termSig zero Vnil .
Fixpoint tS t := @Fun termSig succ (Vcons t Vnil) .
Fixpoint tchoose (t1 t2 : T) : term termSig :=
  @Fun termSig choose (Vcons t1 (Vcons t2 Vnil)) .
Fixpoint tplus (t1 t2 : T) : term termSig :=
  @Fun termSig plus (Vcons t1 (Vcons t2 Vnil)) .
Definition v1 := Var termSig 1 . Definition v2 := Var termSig 2 .
Definition tone := tS tO .
Definition ttwo := tS tone .
Definition tthree := tS ttwo .
```

Finally we can use the terms to construct the rules of our theory. The term constructors O and S are not included in the below rules because they are purely constructors and therefore implicit based on the model defined earlier. Their rules are inherently O $\rightarrow$ O and S V1 $\rightarrow$ S V1. The final rewriting logic theory is then just a list of rewrite rules as all of the rest of components of the tuple are captured in the types.

We actually define two separate theories below, *rwt_nat*, and *rwt_nat_plus*, where the second theory includes the *choose* operations of the first along with rules defining an addition operation. The addition operation makes for some more interesting examples later, while the shorter *rwt_nat* theory is faster when used with automation.

```
Definition rl_choose_left := @mkRule termSig (tchoose v1 v2) v1 .
Definition rl_choose_right := @mkRule termSig (tchoose v1 v2) v2 .
Definition rl_plus_O := @mkRule termSig (tplus v1 tO) v1 .
Definition rl_O_plus := @mkRule termSig (tplus tO v1) v1 .
Definition rl_plus_S_S := @mkRule termSig (tplus (tS v1) (tS v2)) (tS (tS (tplus
v1 v2))) .
Notation "r ':::' sub" := (cons r sub) (at level 80, right associativity) .
Definition rwt_nat := rl_choose_left ::: rl_choose_right ::: nil .
```

```
Definition rwt_nat_plus : (rwl_theory termSig) := rl_plus_O ::: rl_O_plus :::
rl_plus_S_S ::: rl_choose_left ::: rl_choose_right ::: nil .
```

We can now show that the terms O ? 1 → 1 and O ? 1 → O are valid in the theory by creating proof derivations from the rules of the proof theory. In both cases, the proof is by replacement using the appropriate *choose_right* or *choose_left* rule, and then creating a substitution which matches the actual terms.

The proof is far more verbose than its equivalent pen and paper version since we must also prove all of the components of the replacement rule such as the fact that the chosen rule is actually in the theory, and we provide an exact substitution context and then prove that the desired terms can be constructed from the given terms and the substitution. Finally, we must prove that the substitions used to match the RHS and LHS of terms are proveable in the rewrite theory. In this case, they are the same substition, so that portion of the proof is trivial.

```
Example right_choose : proof_theory termSig rwt_nat (tchoose tO tone) tone .
Proof.
  simpl .
  eapply PF_Replacement .
    instantiate (1 := rl_choose_right) . simpl ; auto 6 .
    instantiate (1 := (cons (1, tO) (cons (2, tone) nil))) . simpl . auto .
    instantiate (1 := (cons (1, tO) (cons (2, tone) nil))) . simpl . auto .
    repeat constructor .
Qed .
```

```
Example left_choose : proof_theory termSig rwt_nat (tchoose tO tone) tO .
```

Proof Omitted

```
Qed .
```

We can also simultaneously rewrite within a substitution while rewriting a term with the same replacement rule by proving a different substition for the right-

hand side of the rule, then providing a proof that the two substitutions maintain a provable relationship. This is done inline example below, but can also be done externally as the proof object created in Coq can be used directly in later proofs.

```
Example inner_choose: proof_theory termSig rwt_nat (tchoose (tchoose tO tone)
tO) tone .
Proof .
  simpl .
  eapply PF_Replacement .
    instantiate (1 := rl_choose_left) . simpl ; auto 6 .
    instantiate (1 := (cons (1, (tchoose tO tone)) (cons (2, tO) nil))) ; simpl
; auto .
    instantiate (1 := (cons (1, tone) (cons (2, tO) nil))) ; simpl ; auto .
    eapply PF_ConsSubstitution .
      apply right_choose .
      repeat constructor .
Qed.
```

More examples of rewriting are given in the automation chapter, *Chapter* 5, as the above just starts to scratch the surface of what having an interactive theorem proving environement for rewriting logic in Coq can provide.

CHAPTER 5

Theorem Proving in Rewriting Logic

Now we can expand on the use of Coq as an interactive proof environment for working with rewriting logic theories by creating tactics to automate portions of the proofs done in chapter 4.

5.1 Tactics

Our goal in interative proofs in rewriting logic is primarily to be able to solve queries of the form:

```
exists t', RWT |- t -> t'
```

That is, given a theory and an initial term, what other possible terms can we reach from it? Ideally, we'd also like to be able to write queries such as

```
RWT |- t -> t'
```

where both t and t' are given, and a proof is automatically found if possible. In Chapter 4, we defined a simple rewriting logic theory and stepped through a simple proof session in the example of right choose. Our strategy for the first goal will mimic a simple rewrite engine in its approach. We will attempt to rewrite the given term with each rule, and failing that, attempt to rewrite each subterm with

the given rule set. The process will then repeat from the top level until no new term is found, or a pre-determined number of steps has been run. Since rewriting logic deals with non-terminating theories, a step counter is introduced for practical purposes.

The Coq tactic language includes a matching operator which provides for a backtracking proof search. A statement like

```
match goal with
| [ |- _ ] => reflexivity
| [ |- _ ] => auto
end
```

will first try to match the current goal against the wild card pattern, which always succeeds, and then execute the reflexivity tactic. If that tactic fails to completely solve the goal, it then tries to match the goal against the next pattern, which in this case is again the wildcard pattern, and this time tries the auto tactic, which always succeeds, even if it doesn't make progress. This form of search will feature heavily in the implementation of our tactics.

Our first set of tactics are all designed to manipulate the proof state with new hypothesis that will then be used to attempt to find a proof in rewriting logic. From the given theory, and the initial ground term, we would like to be able to select a single rewrite rule to operate on in. We would also like to generate the substitution necessary to make that rule match our term is possible, and fail early if not. These tactics do not modify the current proof goal, rather they establish useful hypothesis for solving it later.

`Ltac` *pose_rule_n theory n* := `idtac` ;

```
match eval hnf in theory with
| nil ⇒ fail
| cons ?r ?rest ⇒
  match n with
  | O ⇒ pose r
  | S ?n' ⇒ pose_rule_n rest n'
  end
end .

Fixpoint generateSubstitution {Sig} (t1 : term Sig) (t2 : term Sig) : substitution
Sig :=
  match termMatch t1 t2 with
    | None ⇒ nil
    | Some sub ⇒ sub
  end .

Ltac genSubAsHyp :=
  match goal with
    | [ H : (?r : rewrite_rule _) ⊢ proof_theory _ _ ?t1 _ ] ⇒
      let r' := eval unfold H in H in
        pose (generateSubstitution t1 (lhs r')) as sub
  end ; simpl in * .

Ltac pickRHSasHyp :=
  match goal with
    | [ H : (?r : rewrite_rule _),
        Y : (?sub : substitution _) ⊢ proof_theory _ _ ?t1 _ ] ⇒
      let r' := eval unfold H in H in
        let sub' := eval unfold Y in Y in
          pose (applySubstitution sub' (rhs r')) as rhs' ; simpl in rhs'
  end .
```

Our first tactic, *pose_rule_n*, allows us to enrich the hypothesis environment with a specific rule from the theory. It takes a theory and position as parameters, and then poses the corresponding rule from the theory as a hypothesis, or fails if no such rule exists. Then we have *genSubAsHyp*, which attempts to create a substitution to allow the left-hand side of a rule to match the initial term in the proof. This tactic will succeed only when such a substitution can be found, and makes use of a *termMatch* implementation via generateSubstitution. The source for *termMatch* can be found in appendix A. This tactic gets the rewrite rule to use

from the hypothesis environment and expects *pose_rule_n* to be called first.

In addition to the rule and substitution, we also pre-pick the term we are looking for with *pickRHSasHyp* by applying the substitution to the right-hand side of the rule. Another approach would be to have the term we are looking for be filled in automatically during the proving process. Replacement generates a number of proof oblibations which are

- a rule to use
- a proof that the rule being used is in the theory
- a substitution that allows the left-hand side of the rule to match the input term, and a proof of this
- a substitution that allows the right-hand side of the rule to match the resulting term, and a proof of this
- and a proof that the RHS substitution can be proven from the LHS substitution

Each of the above obligations has a number of existential variables, which are unknowns that must be discovered or provided during the proof process. The term we are looking for is one such existential, and here we have chosen to provide it.

```
Ltac instantiateRule :=
match goal with
| [ H : (_ : rewrite_rule ?sig) ⊢ _ ] ⇒
    let r' := eval unfold H in H in
      instantiate (1 := r')
end .
Ltac useRule := instantiateRule ; simpl ; tauto .
```

```
Ltac proveSubstitution := match goal with
| [ H : (_ : substitution _) ⊢ applySubstitution _ ?t1 = ?t2 ] ⇒
    let s' := eval unfold H in H in
    instantiate (1 := s') ; simpl ; reflexivity
end .

Ltac equalSubs := repeat constructor .

Ltac instantiateRHS :=
  match goal with
    | [ H : (?r : term _) ⊢ proof_theory _ _ _ _ ] ⇒
        let r' := eval unfold H in H in
          instantiate (1 := r')
  end .
```

Our next set of tactics use the hypothesis generated from the previous tactics to solve some of the proof obligations generated when finding a new term using the *replacement* rule from our proof theory.

Given that the environment has been enriched with appropriate hypothesis for a rule and a substitution, *useRule* converts the hyopthesis to a form that can be instantiated for the existential variable representing the rule to use. The proof is then a syntactic simple comparison of the rule with each element of the theory which can be handled by the Coq tactic *tauto*. Similarly, *proveSubstitution* instantiates the substitution from the hypothesis into the proof goal, and then simplifies the *applySubstitution* to show equality with the term of interest.

For these tactics, we take the simplifying approach of always using the same substitution on both the left-hand side and right-hand side portions of the rule, so the proof the substitution compatability can be discharged with the *equalSubs* tactic.

```
Ltac one_step_rewrite := repeat match goal with
| [ ⊢ proof_theory _ ?rwt ?t1 ?t2 ] ⇒ eapply PF_Replacement
| [ H : (?r : rewrite_rule _) ⊢ ruleIn _ _ ] ⇒ useRule
| [ ⊢ applySubstitution _ ?t1 = ?t2 ] ⇒ proveSubstitution
```

```
| [ ⊢ proof_theory_substitutions _ _ ?sub _ ] ⇒ equalSubs

| [ ⊢ proof_theory _ _ _ ] ⇒ eapply PF_Congruence
| [ ⊢ proof_theory_list _ _ _] ⇒ constructor
| [ ⊢ proof_theory _ _ ?t ?t ] ⇒ apply PF_Refl
end .
```

Given a proof goal like "proof_theory signature theory t t"', we can solve it for some inputs with the tactic *one_step_rewrite*. Since reflexivity is always an option, this tactic should never fail always providing a least of proof of the existence of the same term via reflexivity. Prior to that, it attemps to rewrite at the top level via replacement using the rule and substitution found in the environment.

```
Ltac try_rules' theory solver :=
  match eval hnf in theory with
    | nil ⇒ idtac
    | cons ?r ?rest ⇒
      try(solve[pose r; genSubAsHyp ; pickRHS ; solver ])
          —— try_rules' rest solver
  end .
Ltac OneStepRewrite :=
  match goal with
  | [ ⊢ proof_theory _ ?rwt _ _ ] ⇒
      try_rules' rwt one_step_rewrite
  end .
```

OneStepRewrite, along with *try_rules'*, set up the hypothesis environment for *one_step_rewrite* choosing each rule in order, and posing a suitable substitution for that rule if one exists. Since Coq tactic backtracking only works withing a match, we need *try_rules'* to be a recursive tactic that attempts to fully solve the goal with each rule, otherwise the proof automation can get stuck with a partially solved goal which is no longer actually solveable.

```
Lemma autoChoose : ∃ t, proof_theory termSig rwt_nat (tchoose tO tone) t .
```

```
Proof .
  eapply ex_intro .
  OneStepRewrite .
Qed.
```

These tactics let use restate our example from Chapter 4 as the lemma *autoChoose* above. Now, instead of supplying *tone* or *tO* as the term we are looking for, we allow the tactics to find a suitable term, along with a proof, automatically.

```
Print autoChoose .
```

```
autoChoose =
ex_intro
  (fun t : term termSig ⇒ proof_theory termSig rwt_nat (tchoose tO tone) t)
  (Fun zero Vnil)
  (let r := rl_choose_left in
   let sub := generateSubstitution (tchoose tO tone) (lhs rl_choose_left) in
   let rhs' :=
     applySubstitution
       ((2, Fun succ (Vcons (Fun zero Vnil) Vnil))
        :: (1, Fun zero Vnil) :: nil) (rhs rl_choose_left) in
   PF_Replacement termSig rwt_nat
     (Fun choose (Vcons (Fun zero Vnil) (Vcons tone Vnil)))
     (Fun zero Vnil) rl_choose_left
     ((2, Fun succ (Vcons (Fun zero Vnil) Vnil)) :: (1, Fun zero Vnil) :: nil)
     ((2, Fun succ (Vcons (Fun zero Vnil) Vnil)) :: (1, Fun zero Vnil) :: nil)
     (or_introl eq_refl) eq_refl eq_refl
     (PF_ReflSub termSig rwt_nat
        ((2, Fun succ (Vcons (Fun zero Vnil) Vnil))
         :: (1, Fun zero Vnil) :: nil)))
     : exists t : term termSig,
         proof_theory termSig rwt_nat (tchoose tO tone) t
```

The tactic has found the term "*Fun zero Vnil*" to be reachable from our original term, and has generated a proof in our proof theory using the replacement rule. The result also includes the specific rule from the theory that was use, the two substitutions, and all of the intermediary Coq proof objects used to generate the final rewriting logic proof.

```
Ltac rewrite_n n :=
match n with
| 0 ⇒ eapply PF_Refl
| S ?n' ⇒ eapply PF_Transitive ; [OneStepRewrite | rewrite_n n']
end ; eapply PF_Refl .
```

We can now wrap up our tactic to run through multiple steps as in *rewrite_n*. These tactics now roughly correspond to a simple rewrite engine, but generate actual Coq proofs corresponding to proofs in our proof theory for rewriting logic. Thus Coq now serves both an interactive theorem proving environment for rewriting logic, and also as an automated rewriting engine for our proof theory. While these tactics have the same limitations of the proof engine in that they choose a specific strategy for directing the rewriting, the interactive environment allows us to manually specify portions of the proof as necessary, while still using the automation for other portions, to provide a more comprehensive environment for evaluating rewriting logic theories.

CHAPTER 6

Conclusions and Future Work

The original, category theoretical model for rewriting logic was designed in part to answer the question, "When are two concurrent computations equivalent?" For this, we must be able to make a distinction between different proof theoretic derivations when we would like for them to be considered the same computation.

6.1 Future Work

From the verified foundations, we will focus on a few areas of exploration.

6.1.1 Equational Sublogic

One common theme in the definition and implementations of rewriting logic is the use of equational sublogics for defining static parts of a theory, while the rewriting rules can be considered the dynamic parts.

From a theoretical perspective, equational logic was an earlier model for term rewriting, where theories were considered to be replacing equals with equals, instead of defining transitions between states. From an implementation perspective, the handling of equations and rewrite rules can both be implemented as a term replacement operation, with the only difference being when such a term replacement is a valid operation.

For a example specification that we'd prefer to be an equational specification rather a rewriting logic specification we turn back to the formalization of peano nats in Maude.

```
Example in Maude:
  fmod Nat .
    sort ANY .
    op 0 : → ANY .
    op S : ANY → ANY .
    op + : ANY ANY → ANY .
    vars N M : ANY .
    eq 0 + N = N .
    eq N + 0 = N .
    eq S(N) + S(M) = S(S(N + M)) .
  endfm
```

The specification of addition is both terminating and normalizing. Any time we see an addition term, we'd generally like to reduce it as far as possible as the intermediary terms are rarely, if ever, interesting. Our current definition of rewriting logic does not allow this, and therefore in a theory like this, we will often have to deal with many uninteresting terms and states. If we define an equational sublogic and require certain properties for the equational theories, we can fully reduce such terms and deal only with rewriting logic terms using normalized vales from the equational theory.

To enable the desired automation, we require that the equational theory be terminating and confluent. This is a property that is required, but not checked, in all existing rewriting logic implementations with an equational sublogic. A further requirement is that the equational rules and the rewriting rules should be coherent, which means that rewriting via the equations or the rewrite rules is not order dependent and will lead to the same outcomes. This allows for a more efficient implementation by first normalizing terms via the equational rules before applying the rewriting rules to shrink set of terms.

The equational sublogic is accounted for in other models of rewriting logic by defining the logic not over terms, but rather over equivalence classes of terms

where the equivalence class is defined via the equational theory. We can allow for a similar formalization by replacing the Equiv model relationship in our term algebra (Subsection 3.2.2) with a formalization of equality via an equational theory. Without further change to our model, this should allow for the specification of rewriting modulo equations.

We can then go further and attempt to formalize the requirements that an equational theory be both terminating and confluent by requiring that proofs of these properties be supplied along with the equational theory. Additionally, we can require a proof of coherence between the equational theory and the rewriting logic theory to move towards a verified implementation of rewriting modulo equations.

Since these properties are currently implicit in other system, making them explicit can help to reduce errors that can be encountered in rewriting logic theories where the proper verifications weren't made. ¡¿ project is attempting to accomplish a similar goal by integrating the sufficient complete checker, coherence check, and some of other Maude based tools we discussed in Section 1.4 into a single, unified platform.

6.1.2 Verified Extracted Implementation

Along the vein of tools for working with rewriting logic theories, we can use the rewriting logic specification to create a verified implementation of rewriting logic. We could start by defining a rewrite function whose return type is *t' : term Sig | derives Sig rwl t t'*. This return type expresses that it returns a term derivable from the original term in the given rewrite theory and provides a proof of the derivation in our semantics.

When the implementation of this function type checks in Coq, we have a proof that

this function only generates provably derivable terms according to the semantics of rewriting logic. We can then extract then function from Coq into another programming language, such as Ocaml, where all of the proof terms are stripped out leaving behind just an implementation of rewriting logic that has been proven correct.

6.1.3 Quotient Relationships

In Subsection 3.2.2 we discussed the quotienting relation as the means behind semantics of the logic. However, we only provided one quotient relationship implementation. The definition of additional quotient relations, such as a behavioral equivalence where any two proof terms are considered equal if they have the same initial and final terms are a natural follow on to this initial research.

```
Definition behavioral_eq (t1 t2 : model_type ts rwt) :=
  match t1,t2 with
    | existT (X1, Y1) pf, existT (X2, Y2) pf2 ⇒
      eq X1 X2 ∧ eq Y1 Y2
  end .
```

We can define a property, *behavioral_eq*, which is supposed to be an equality relationship over elements of a model. To allow this to function as an equality relationship in Coq, we first need to prove that this relation is reflexive, symmetric and transitive. Afterwards, we can define a quotient algebra with behavioral equivalence as the quotienting relationship instead of syntactic equality.

6.1.4 Type System

Multi-sorted, order-sorted, and potentially deeper type systems.

6.2 Conclusion

We have now defined a new semantics for rewriting logic, and have shown how by formalizing this semantics in Coq, we are able to create a theorem proving environment for working within rewriting logic. Additionally, by defining both a model theoretic semantics and a proof theoretic semantics within Coq, we were able to create a machine verifiable proof that rewriting logic is both sound and complete. From here, we have a platform for a variety of work both in and on rewriting logic, and have outlined some of the potential next steps.

LIST OF REFERENCES

[1] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical Computer Science*, vol. 96, pp. 73–155, April 1992. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(92)90182-F

[2] T. Coq Development Team, *The Coq Reference Manual, version 8.4*, Aug. 2012, available electronically at http://coq.inria.fr/doc.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, Eds., *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, ser. Lecture Notes in Computer Science, vol. 4350. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek, "Elan: A logical framework based on computational systems." *Electronic Notes in Theoretical Computer Science*, 1996.

[5] R. Diaconescu and K. Futatsugi, "Logical foundations of cafeobj," *Theoretical Computer Science*, vol. 285, no. 2, pp. 289–318, 2002.

[6] F. Durán, "Reflective module algebra with applications to the maude language," Ph.D. dissertation, University of Málaga, 1999.

[7] C. Kirchner, H. Kirchner, and P. etienne Moreau, "Elan from a rewriting logic point of view," *Theoretical Computer Science*, p. 2002, 2002.

[8] F. Durán and J. Meseguer, "A church-rosser checker tool for conditional order-sorted equational maude specifications," in *Rewriting Logic and Its Applications*, ser. Lecture Notes in Computer Science, P. Ölveczky, Ed. Springer Berlin / Heidelberg, 2010, vol. 6381, pp. 69–85.

[9] F. Durán and J. Meseguer, "A maude coherence checker tool for conditional order-sorted rewrite theories," in *Proceedings of the 8th international conference on Rewriting logic and its applications*, ser. WRLA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 86–103. [Online]. Available: http://dl.acm.org/citation.cfm?id=1927806.1927816

[10] F. Durán, C. Rocha, and J. M. Álvarez, "Towards a maude formal environment," in *Formal Modeling: Actors, Open Systems, Biological Systems*, ser. Lecture Notes in Computer Science. Springer, 2011, vol. 7000, pp. 329–351.

[11] F. Blanqui and A. Koprowski, "Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates." *Mathematical Structures in Computer Science*, vol. 21, pp. 827–859, 2011.

[12] F. Honsell, M. Miculan, and I. Scagnetto, "pi-calculus in (co)inductive type theory," *Theoretical Computer Science*, vol. 253, p. 2001, 2001.

[13] V. Capretta. "Equational reasoning in type theory." May 2000. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.9448

[14] V. Capretta, "Universal algebra in type theory," in *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, ser. Lecture Notes in Computer Science (LNCS), Y. Bertot, G. Dowek, A. Hirschowits, C. Paulin, and L. Théry, Eds., vol. 1690. Springer, 1999, pp. 131–148.

[15] T. Ramsamujh, "Equational logic and abstract algebra," in *Proceedings: Thirty-fourth Annual Meeting of the Florida Section of the Mathematical Association of America*, D. Kerr and B. Rush, Eds., April 2002. [Online]. Available: http://sections.maa.org/florida/proceedings/2001/default.htm

[16] S. Coupet-Grimal, "An axiomatization of linear temporal logic in the calculus of inductive constructions," *The Journal of Logic and Computation*, vol. 13, pp. 801–813, 2002.

[17] N. Paiva, "Temporal logic in coq," Master's thesis, Instituto Superior Técnico, 1999.

[18] R. Affeldt and N. Kobayashi, "A coq library for verification of concurrent programs," *Electron. Notes Theor. Comput. Sci.*, vol. 199, pp. 17–32, Feb. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2007.11.010

[19] F. Honsell, M. Miculan, and I. Scagnetto, "pi-calculus in (co)inductive type theory," *Theoretical Computer Science*, vol. 253, p. 2001, 2001.

[20] G. Huet and D. C. Oppen, "Equations and rewrite rules: A survey," in *Formal Language Theory: Perspectives and Open Problems*. Academic Press, 1980.

[21] N. Dershowitz and J.-P. Jouannaud, "Rewrite systems," in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 1990, pp. 243–320.

[22] Terese, Ed., *Term Rewriting Systems*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003, vol. 55.

[23] V. Manca and A. Salibra, "Soundness and completeness of the birkhoff equational calculus for many-sorted algebras with possibly empty carrier sets," *Theoretical computer science*, vol. 94, no. 1, pp. 101–124, 1992.

[24] W. Wechler, *Universal Algebra for Computer Scientists (EaTCS Monographs on Theoretical Computer Science)*. Springer-Verlag, 1992.

[25] H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*, ser. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985, vol. 6.

[26] T. Coquand, "Metamathematical investigations of a calculus of constructions," INRIA, Tech. Rep. RR-1088, Sept. 1989. [Online]. Available: http://hal.inria.fr/inria-00075471/en/

[27] T. Coquand and G. Huet, "The calculus of constructions," *Inf. Comput.*, vol. 76, pp. 95–120, February 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=47724.47725

[28] A. Chlipala, *Certified Programming with Dependent Types*. MIT Press, 2013. [Online]. Available: http://adam.chlipala.net/cpdt/

[29] S. Thompson, *Type theory and functional programming*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1991.

[30] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

APPENDIX

Additional Source Code

A.1 Utilities

```
Require Export List .
Require Export ListSet .
Require Export String .
Require Import Signature .
Require Import substitution .
Set Implicit Arguments .

Open Scope string_scope .

Definition lst1 := cons "1a" (cons "2a" (cons "3a" nil)) .
Definition lst2 := cons "1b" (cons "2b" (cons "3b" nil)) .

Eval simpl in list_prod lst1 lst2 .

Definition flat_prod (A : Type) (item : A * A) :=
  match item with
    | (v1, v2) ⇒ cons v1 (cons v2 nil)
  end .

Definition flat_prod2 (A : Type) (item : A * (list A)) :=
  match item with
    | (v1, v2) ⇒ cons v1 v2
  end .

Check flat_prod .
Check flat_prod2 .

Fixpoint testloop (A : Type) ( lst : list (A * A) ) : list (list A) :=
  match lst with
    | nil ⇒ nil
    | cons h rst ⇒
      cons (flat_prod h) (testloop rst)
  end .

Fixpoint prod_list (A : Type) ( lst : list (A * list A) ) : list (list A) :=
  match lst with
    | nil ⇒ nil
    | cons h rst ⇒
```

```
      cons (flat_prod2 h) (prod_list rst)
  end .

Eval simpl in testloop (list_prod lst1 lst2) .

Fixpoint nil_list (A : Type) (n : nat) : list (list A) :=
  match n with
    | 0 ⇒ nil
    | S n' ⇒ cons nil (nil_list A n')
  end .

Eval simpl in nil_list string 3 .
Check nil_list string 3 .
Definition lst_lst := nil_list string 3 .
(*Eval simpl in list_prod lst2 (cons nil nil) .*)

Definition prep (A : Type) (lst : list A) (p : list (list A)) :=
  list_prod lst p .
Check prep .
Eval simpl in prep lst2 (cons nil nil) .

Eval simpl in let tmp := prep lst2 (cons nil nil) in
              prod_list tmp .

Definition i1 := prod_list (prep lst2 (cons nil nil)) .

(* Below returns the desired [ [1a, 2a], [1a, 2b] ... [1c, 3b], [1c, 3c] ] *)
Eval simpl in prod_list (list_prod lst1 i1) .

(*
        Error: The term "flat_prod" has type "forall A : Type, A * A →
list A"
  while it is expected to have type "?31 → ?32".
  *)
 (*
 Eval simpl in map flat_prod (list_prod lst1 lst2) .
 *)

 (* combos:
    list functions takes a list of lists representing the possible transitions
  for each element in a term list. It then combines all possible permutations
  into a single list of lists, where each sublist represents one possible
  transition path for the original term list

  ie [ [1a, 2a, 3a] [1b, 2b] [1c, 2c, 3c] ]
  ⇒
```

```
   [ [1a, 1b, 1c], [1a, 1b, 2c], ..., [3a, 2b, 3c] ]
 *)
  Fixpoint combos (A : Type) (lsts : list (list A)) : list (list A) :=
  match lsts with
    | nil ⇒ nil
    (* below is currently necessary because list_prod
       of lst nil ⟹nil instead of returning a list of lists *)
    | cons h nil ⇒ prod_list (prep h (cons nil nil))
    | cons h rst ⇒
      let tails := combos rst in
      prod_list (list_prod h tails)
  end .

Eval simpl in combos (cons lst1 (cons lst2 nil)) .

(*****************************************************************)

(* Convert a list of vals to a list of lists of those vals.
   Essentially a powerset of combinations of those values /
   positions.

   ex. [1, 2, 3] ⇒ [[1], [2], [3], [1, 2], [2, 3], [1, 3], [1,2,3], []]
*)

Fixpoint list_to_powerset (A : Type) (lst : list A) :
   list (list A) :=
match lst with
| nil ⇒ nil
| cons h rst ⇒ nil
end .

(*****************************************************************)

(* listToTriples breaks a list of elements into a list of triples
   representing all possible split points in the list and the
   items before and after the split point

   ex listToTriple [1, 2, 3] ⇒
    [ ([], 1, [2, 3]), ([1], 2, [3]), ([1, 2], 3, []) ]
  *)

Fixpoint listToTriples A (lst : list A) :=
let fix aux F R :=
    match R with
      | nil ⇒ nil
      | cons H R'
```

```
        ⇒ cons (F, H, R')  (aux (app F (cons H nil)) R')
    end
in aux nil lst.

Eval simpl in listToTriples (cons 1 (cons 2 (cons 3 nil))) .

(*************************************************************)

(* Convert a list of options to an option containing a list.
   Any 'None' in the original list results in a None
   being returned.
   If the original list is all "Some's", then a Some lst
   is returned

   Note: option_map2 could be rewritten to be
    extractOptionList (map fnc lst)
 *)
Fixpoint extractOptionList (A : Type) (lst : list (option A)) : option (list A) :=
  let fix loop lst accum :=
      match lst with
        | nil ⇒ Some (rev accum)
        | cons h rest ⇒
            match h with
              | Some val ⇒ loop rest (cons val accum)
              | None ⇒ None
            end
      end
  in loop lst nil .

Fixpoint toList A n (v : vector A n) : list A :=
match v with
| Vnil ⇒ nil
| Vcons h _ rest ⇒ cons h (toList rest)
end .

(*
Fixpoint extractOptionVector (A : Type) (n : nat) (v : vector (option A) n)
  : option (vector A _)
:=
  let lst := extractOptionList (to_list v) in
  match lst with
  | None ⇒ None
  | Some l' ⇒ Some (of_list l')
  end .

Check extractOptionVector .
```

```
  let fix loop lst accum :=
      match lst with
        | Vnil ⇒ Some (rev accum)
        | Vcons h _ rest ⇒
             match h with
               | Some val ⇒ loop rest (cons val accum)
               | None ⇒ None
             end
      end
  in loop v nil .
*)

(*************************************************************)

(*
Fixpoint map_maybe (A : Type) (B : Type)
               (fnc : A → option B) (lst : list A) :
  option (list B) :=
  let fix loop lst accum :=
      match lst with
        | nil ⇒ Some accum
        | cons g rest ⇒
            let tmp := fnc g in
              match tmp with
                | Some val ⇒ loop rest (cons val accum)
                | None ⇒ None
              end
      end
  in loop lst nil .
*)

(*************************************************************)

(* begin hide *)

Section Utils .
  Variable sgn : termSignature .

(* The empty_substitution is frequently used so we define it
 * here for convenience.
 *)
Definition empty_substitution : substitution sgn := nil .
(* end hide *)

(* begin hide *)
```

```
(*
  We also define some utility functions for working on and with
substitutions that will be used later. The update function merely prepends
a new mapping pair to the head of the substitution without checking to see whether or
not the variable is already bound. In situtation where it may be already bound
and thus be incorrect, XXXX is responsible for ensuring the variable isn't
already bound.
  TODO - what's XXXX? ie, what's reponsible ?

 *)
Notation variable := nat (only parsing).

Definition update (sigma : substitution sgn) (X : variable) (t : term sgn) : substitution sgn :=
  cons (X, t) sigma .

(*
  Next we define a function to apply a substitution to a term.
Currently, we don't allow this function to fail. If some variable are
unbound in the substitution, they remain as variables in the resultant term.
In a similar vein, extra bound variable in a substitution are simply ignored.

TODO: should applySubstitution be able to fail and disallow unbound vars?
TODO: are we only allowing rewriting of ground terms?
 *)
Fixpoint lookup (st : substitution sgn) (i : variable) : option (term sgn) :=
  match st with
    | nil ⇒ None
    | cons (id, t') rest ⇒ if beq_nat id i then Some t' else lookup rest i
  end .

Require Import VecUtil .

Fixpoint applySubstitution (st : substitution sgn) (t : term sgn) : term sgn :=
match t with
| Var x ⇒ match lookup st x with
           | None ⇒ t
           | Some t' ⇒ t'
          end
| Fun f tls ⇒ Fun sgn f (Vmap _ _ (applySubstitution st) _ tls)
end.

End Utils .
(* end hide *)
```

A.2 Tactic Utilities

```
Require Import Signature .
```

```
Require Import ProofTheory .
Require Import substitution .
Require Import rwlNatPlus .

Section rewriteUtilities .

(*
 If a variable occurs multiple times, it could be bound differently, and the
result applySubstitution will fail.
 -- NOTE: this isn't actually true. The checkSub call will cause termMatch to
  fail is a variable is bound twice to a non-identical value, so there might
  be multiple bindings for a var, but they should be identical.
 -- NOTE: If T1 contains a variable, term match fails.
*)
(* begin hide *)
Fixpoint checkSingle {Sig} n (t : term Sig) (s : substitution Sig) : bool :=
match s with
| nil ⇒ true
| cons (n', t') rest ⇒ if beq_nat n n' then ·beq_term Sig t t' else checkSingle n t rest
end .

Fixpoint checkSub {Sig} (s1 s2: substitution Sig) : bool :=
match s1 with
| nil ⇒ true
| cons (n, t) rest ⇒ if checkSingle n t s2 then checkSub rest s2 else false
end .
(* end hide *)

Fixpoint termMatch {Sig} (T1 : term Sig) (T2 : term Sig) : option (substitution Sig) :=
match (T1, T2) with
| (Fun F P1, Var V1) ⇒ Some (cons (V1, T1) nil)
| (Fun F P1, Fun F' P2) ⇒ if negb (beq_term_symb Sig F F') then None else
  let fix aux {Sig n m} (L1 : vector (term Sig) n) (L2 : vector (term Sig) m) iSub :=
      match (L1, L2) with
          | (Vnil, Vnil) ⇒ Some iSub
          | (Vcons t _ r, Vnil) ⇒ None (* mismatched param list lens *)
          | (Vnil, Vcons t _ r) ⇒ None (* mismatched param list lens *)
          | (Vcons t _ r, Vcons t' _ r') ⇒ let innerMatch := termMatch t t'
            in match innerMatch with
                | None ⇒ None
                | Some sub ⇒ if checkSub sub iSub
                             then aux r r' (app sub iSub)
                             else None
            end
      end
   in aux P1 P2 nil
```

```
| (Var V1, _) ⇒ None
end .

End rewriteUtilities .
```

A.3 Equivalence Proof

```
(* begin hide *)
Require Import SfLib .
Require Import Signature .
Require Import ProofTheory .
Require Import Model .

(*
  To prove the congruence case below, we need to have
    reachbility Sig RWT t1 t2 → models Sig RWT t1 t2 in the hypothesis
    for the M_Cons case, but it's not there. How do we get it there?
    Using lemma?
  -- Answer was mutual induction.
*)

Check proof_theory_ind .

(* end hide *)

Theorem rwl_sound : forall (Sig : termSignature)
                           (rwt : rwl_theory Sig)
                           (t t' : term Sig),
    proof_theory Sig rwt t t' → mSat Sig rwt t t' .
Proof .
intro Sig .
  (* Proof by induction over the structure of the proof_theory proof *)
  apply (proof_theory_mut Sig
   (fun rwt ⇒ fun t ⇒ fun t' ⇒
      mSat Sig rwt t t')
   (fun n ⇒ fun rwt ⇒ fun t ⇒ fun t' ⇒
      lmSat Sig rwt t t')
   (fun rwt ⇒ fun sub ⇒ fun sub' ⇒
      subSat Sig rwt sub sub')
   ) ; unfold mSat .
Case "Reflexivity" .
intros t1 RWT . exists (M_Refl Sig t1 RWT) . tauto .
Case "Congruence" .
intros RWT f tl tl2 H0 H1 .
  unfold lmSat in H1 .
  destruct H1 .
  exists (M_Congruence Sig RWT f tl tl2 x) .
```

```
  auto .
Case "Weak" .
intros rule RWT t t' pf H .
  destruct H .
  exists (M_Weak Sig rule RWT t t' x) .
  auto .
Case "Replacement".
intros RWT inputTerm outputTerm rule Subs Subs' rIn lhsEq rhsEq subRed subSatObj .
  unfold subSat in subSatObj .
  destruct subSatObj .
  exists (M_Replacement Sig RWT inputTerm outputTerm rule Subs Subs' rIn lhsEq rhsEq x) .
  auto .
Case "Transitivity" .
intros t1 t2 t3 RWT r12 e12 r23 e23 .
  destruct e12 . destruct e23 .
  exists (M_Transitive Sig t1 t2 t3 RWT x x0) .
  auto .
Case "List Nil" .
intros . exists (M_Nil Sig RWT) . auto .
Case "List Cons" .
intros .
  destruct H0 .
  unfold lmSat in H2 ; destruct H2 .
  exists (M_Cons Sig RWT m rest1 rest2 t1 t2 x x0) ; auto .
Case "Substitution Reflexive" .
intros . unfold subSat . exists (M_ReflSub Sig RWT st) . auto .
Case "Substitution HeavyLeft" .
intros . unfold subSat in *. destruct H0 .
  exists (M_HeavyLeft Sig RWT entry sub sub' x) . auto .
Case "Substutition Cons" .
intros . unfold subSat .
  destruct H0 .
  unfold subSat in H2 . destruct H2 .
  exists (M_ConsSubstitution Sig RWT id1 t1 t2 rest1 rest2 x x0 ).
  auto .
Qed.

Scheme models_mut := Minimality for models Sort Prop
with lmodels_mut := Minimality for lmodels Sort Prop
with subsModel_mut := Minimality for subsModel Sort Prop .

Theorem rwl_complete :
  forall (Sig : termSignature) (rwt : rwl_theory Sig) (t t' : term Sig),
    mSat Sig rwt t t' →
    proof_theory Sig rwt t t' .
intro Sig .
```

```
unfold mSat . destruct 1 .
(* Proof by induction over the structure of the generators proof *)
apply (models_mut Sig
   (fun rwt ⇒ fun t ⇒ fun t' ⇒
      proof_theory Sig rwt t t')
   (fun n ⇒ fun rwt ⇒ fun t ⇒ fun t' ⇒
      proof_theory_list Sig _ rwt t t')
   (fun rwt ⇒ fun sub ⇒ fun sub' ⇒
      proof_theory_substitutions Sig rwt sub sub')
   ) .
Case "Identity" .
intros t1 RWT . apply PF_Refl .
Case "Sigma-structure" .
intros .  apply PF_Congruence ; assumption .
Case "Weak" .
intros . apply PF_Weak ; assumption .
Case "Replacement" .
intros RWT inputTerm outputTerm rule Subs Subs' rIn
       lhsEq rhsEq subsMod subsReach .
  apply (PF_Replacement _ _ _ _ rule Subs Subs') ; assumption .
Case "Transitivity" .
intros t1 t2 t3 RWT H1 H2 H3 . apply PF_Transitive ; assumption .
Case "List Nil" .
intros . apply PF_Nil .
Case "List Cons" .
intros . apply PF_Cons ; assumption .
Case "Substitution Reflexive" .
intros . apply PF_ReflSub ; assumption .
Case "Substitution HeavyLeft" .
intros . apply PF_HeavyLeft ; assumption .
Case "Substitution Cons" .
intros . apply PF_ConsSubstitution ; assumption .
assumption .
Qed.

Theorem rwl_sound_and_complete :
  forall (Sig : termSignature) (rwt : rwl_theory Sig) (t t' : term Sig),
    proof_theory Sig rwt t t' ↔
    mSat Sig rwt t t' .
split .
apply rwl_sound .
apply rwl_complete .
Qed .
```

BIBLIOGRAPHY

Affeldt, R. and Kobayashi, N., "A coq library for verification of concurrent programs," *Electron. Notes Theor. Comput. Sci.*, vol. 199, pp. 17–32, Feb. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2007.11.010

Bertot, Y. and Castéran, P., *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, ser. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

Blanqui, F. and Koprowski, A., "Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates." *Mathematical Structures in Computer Science*, vol. 21, pp. 827–859, 2011.

Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E., and Vittek, M., "Elan: A logical framework based on computational systems." *Electronic Notes in Theoretical Computer Science*, 1996.

Capretta, V., "Universal algebra in type theory," in *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, ser. Lecture Notes in Computer Science (LNCS), Bertot, Y., Dowek, G., Hirschowits, A., Paulin, C., and Théry, L., Eds., vol. 1690. Springer, 1999, pp. 131–148.

Capretta, V. "Equational reasoning in type theory." May 2000. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.21.9448

Chlipala, A., *Certified Programming with Dependent Types*. MIT Press, 2013. [Online]. Available: http://adam.chlipala.net/cpdt/

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. L., Eds., *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, ser. Lecture Notes in Computer Science, vol. 4350. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

Coq Development Team, T., *The Coq Reference Manual, version 8.4*, Aug. 2012, available electronically at http://coq.inria.fr/doc.

Coquand, T., "Metamathematical investigations of a calculus of constructions," INRIA, Tech. Rep. RR-1088, Sept. 1989. [Online]. Available: http://hal.inria.fr/inria-00075471/en/

Coquand, T. and Huet, G., "The calculus of constructions," *Inf. Comput.*, vol. 76, pp. 95–120, February 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=47724.47725

Coupet-Grimal, S., "An axiomatization of linear temporal logic in the calculus of inductive constructions," *The Journal of Logic and Computation*, vol. 13, pp. 801–813, 2002.

Dershowitz, N. and Jouannaud, J.-P., "Rewrite systems," in *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 1990, pp. 243–320.

Diaconescu, R. and Futatsugi, K., "Logical foundations of cafeobj," *Theoretical Computer Science*, vol. 285, no. 2, pp. 289–318, 2002.

Durán, F., "Reflective module algebra with applications to the maude language," Ph.D. dissertation, University of Málaga, 1999.

Durán, F. and Meseguer, J., "A church-rosser checker tool for conditional order-sorted equational maude specifications," in *Rewriting Logic and Its Applications*, ser. Lecture Notes in Computer Science, Ölveczky, P., Ed. Springer Berlin / Heidelberg, 2010, vol. 6381, pp. 69–85.

Durán, F. and Meseguer, J., "A maude coherence checker tool for conditional order-sorted rewrite theories," in *Proceedings of the 8th international conference on Rewriting logic and its applications*, ser. WRLA'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 86–103. [Online]. Available: http://dl.acm.org/citation.cfm?id=1927806.1927816

Durán, F., Rocha, C., and Álvarez, J. M., "Towards a maude formal environment," in *Formal Modeling: Actors, Open Systems, Biological Systems*, ser. Lecture Notes in Computer Science. Springer, 2011, vol. 7000, pp. 329–351.

Ehrig, H. and Mahr, B., *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*, ser. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985, vol. 6.

Honsell, F., Miculan, M., and Scagnetto, I., "pi-calculus in (co)inductive type theory," *Theoretical Computer Science*, vol. 253, p. 2001, 2001.

Honsell, F., Miculan, M., and Scagnetto, I., "pi-calculus in (co)inductive type theory," *Theoretical Computer Science*, vol. 253, p. 2001, 2001.

Huet, G. and Oppen, D. C., "Equations and rewrite rules: A survey," in *Formal Language Theory: Perspectives and Open Problems*. Academic Press, 1980.

Kirchner, C., Kirchner, H., and etienne Moreau, P., "Elan from a rewriting logic point of view," *Theoretical Computer Science*, p. 2002, 2002.

Manca, V. and Salibra, A., "Soundness and completeness of the birkhoff equational calculus for many-sorted algebras with possibly empty carrier sets," *Theoretical computer science*, vol. 94, no. 1, pp. 101–124, 1992.

Meseguer, J., "Conditional rewriting logic as a unified model of concurrency," *Theoretical Computer Science*, vol. 96, pp. 73–155, April 1992. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(92)90182-F

Paiva, N., "Temporal logic in coq," Master's thesis, Instituto Superior Técnico, 1999.

Ramsamujh, T., "Equational logic and abstract algebra," in *Proceedings: Thirty-fourth Annual Meeting of the Florida Section of the Mathematical Association of America*, Kerr, D. and Rush, B., Eds., April 2002. [Online]. Available: http://sections.maa.org/florida/proceedings/2001/default.htm

Terese, Ed., *Term Rewriting Systems*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003, vol. 55.

Thompson, S., *Type theory and functional programming*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1991.

Wechler, W., *Universal Algebra for Computer Scientists (EaTCS Monographs on Theoretical Computer Science)*. Springer-Verlag, 1992.