Spring 2021

A Unification Algorithm For The First Order Theory of Quandles

Elliott N. Goldstein
Bard College, eg9699@bard.edu

A Unification Algorithm For The First Order Theory of Quandles

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Elliott Goldstein

Annandale-on-Hudson, New York
May, 2021

Abstract

The long-range goal of this project is to develop an algorithm to decide whether two terms are unifiable over the theory of quandles. First, it is shown that the general E-unification reduces to the E-matching problem due to the right-cancellation axioms of quandles. The E-matching process takes the general narrowing approach to equational matching. However, a naive application of narrowing is, at best, recursively enumerable and hence will not terminate given terms that do not match. This modification of narrowing places a hard limit on the use of the delta rules of the term rewriting system for quandles to ensure termination. It is implemented in the SWI-Prolog logic programming language. The question remains open as to whether the imposed limits still allow the program to find a unifier for all matching pairs.

Contents

Acknowledgments

Writing this paper has been no small task, and truly has taken me from September to May, working all the way through winter break, for it to reach its final form. I'd like to thank my advisor Bob McGrail for his great depth of knowledge of quandles, rewriting systems, knot theory, and Prolog. I'd like to wish Bob good luck with Mac Jones as the future of the Patriots. Be patient with the young QB's, not everyone can be Brady! I'd also like to extend thanks to Keith O'Hara, Sven Anderson, and Keri-Ann Norton, who have provided me with the fundamentals for my entire understanding of computer science.

Dedication

This paper is dedicated to first and foremost, my family- Alan, Debbie, Nina and Lila, who have made me laugh and been an invaluable support system throughout my last 4 years of school. I'd also like to dedicate this paper to my roommate and very close friend Jordan Myers, who watched many hours of Last Chance U with me in the background while we did edits on our papers. Let's hope that our advisors "GONNA FEEL ME" when reading our papers. Additional shoutout to the DK X NAWF group as a whole (IFBB Pro Mike Hannan, JoeBar, and Furphy Moss), as well as the Bard Baseball team, who have provided me with a much needed distraction throughout the course of this paper so I could stay sane.

1

Introduction

The topic of this paper will be the derivation of a unification algorithm that finds a most general unifier for the first order theory of quandles. The term rewriting system discussed in the paper, and it's respective properties, was established in McGrail's paper on the rewrite system for the pure equational theory of quandles [4]. The topic of decidability and it's relation to quandles was discussed in McGrail's and Belk's paper on the word problem [6]. The focus of chapter 2 will be formulating the link between knot theory and equational axioms, building on work from Reidemeister[7] and Rolfsen[2]. In chapter 3, the process of syntactic unification is described, while chapter 4 focuses on the process of equational unification, referencing work from McGrail[4] and Johann[10]. The discussion in chapter 5 is on the properties of termination, confluence, and convergence, referencing Bader and Nipkow's book[14].The next two chapters discuss the usages of the Maude tool and Prolog programming language respectively. In chapter 8, both the motivations and methods for the implementation of a metavariable sytem in Prolog to gain more control over the unification process are discussed. Once the basis of the metavariable system has been established, chapter 9 and chapter 10 provide the pseudocode and formal Prolog implementation for the unification algorithm.

2

Knots and Quandles

A knot is the continuous projection of the unit circle into 3-dimensional space. However, it is often easier to visualize knots as 2-dimensional projections of 3-dimensional knots. A simple example of a knot in 2-d space is Figure 2.0.1, which contains 3 "crossings". This figure is known in knot theory as the trefoil. A crossing is a location of a knot in which the knot crosses over itself. Any knot that realizes it's respective minimal number of crossings is considered reduced.

The algebraic conception of reduction ensures that any term that goes through a "reduction" step is now less complex than it's original form, while the knot conception of reduction states it is impossible to infinitely reduce the number of crossings in a given knot[1]. Intuitively, this is because eventually any knot will become fully untied. A *strand*

Figure 2.0.1. Trefoil

in a knot diagram is a continuous section of a knot that spans the length of a crossing [2], and an *arc* is an unbroken part of a strand through the perspective of a 2D projection of the knot [2]. Furthermore, the axioms contained in the quandle will utilize only the binary

operators / and *. These operators, which may appear to be the operators for division and multiplication, can best be understood simply as operators whose rules are defined by the context of the axioms and quandle. The operators will often be referred to in the paper as simply *mult and div*.

2.1 Ambient Isotopy and Reidemeister Moves

Reidemeister discovered three fundamental "moves", or ways of modifying knots, that can change the number of crossings and structure of a knot [7]. These moves allow to check for if two knots are *ambient isotopic*. Two knots are ambient isotopic if there exists a sequence of Reidemeister moves that transforms one knot to the other. Discovering the ambient isotopy of two knots is equivalent to declaring that one knot is the "deformation" of the other through the Reidemeister moves. In the quandle constructed from the Reidemeister moves, the "$*$" binary operator corresponds to a crossing to the left, and the "/" operator corresponds to a crossing to the right.

2.2 Type 1 Reidemeister Moves

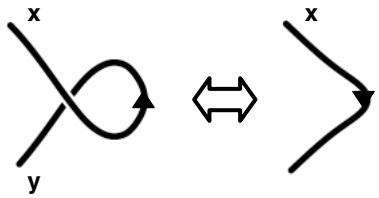


Figure 2.2.1. A Type 1 Reidemeister Move

The first Reidemeister Move corresponds to the theorem of *idempotence*. In Figure 2.2.1, it can be seen that the knot on the left hand side consists of just one strand that consists of the arcs x and y. Arc y is formed by strand x crossing over itself to the right. By the

logic of the crossing algebra, this corresponds to:

$$X * X = Y \tag{2.2.1}$$

However, since the entry point labeled y on the left hand side of Figure 2.2.1 corresponds to the arc labeled x on the right hand side, this Reidemeister move states that y=x. By transitivity, idempotence, in the knot quandle corresponds to the equations

$$X * X = X \tag{2.2.2}$$

$$X/X = X \tag{2.2.3}$$

. Idempotence in knot theory refers to the concept that some simple knot that contains one crossing, when "untied", is simply the same knot just in an untied form. Algebraically, idempotence refers to the phenomenon in which any operator applied to arguments of equal value return a value equal to the arguments to which the operator was applied. The final equation is expressed with uppercase variables as a harbinger of the use of the Prolog programming language later.

2.3 Type 2 Reidemeister Moves

The transformation in Figure 2.3.1 is the Type 2 Reidemeister move. The right-hand side

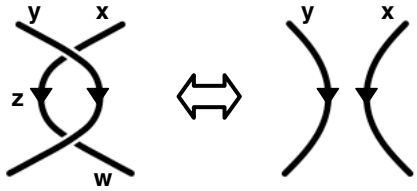


Figure 2.3.1. A Type 2 Reidemeister Move

of Figure 2.3.1 shows two arcs x and y, that, when crossed over each other, produce the

knot on the left hand side of Figure 2.3.1 that contains two crossings. The knot segment on the left-hand side contains the arcs x,y,z and w. Arc z is formed by arc x crossing over arc y to the right, yielding the crossing equation x / y = z. Arc w is formed by arc z crossing over arc y to the left, yielding the crossing algebra z ∗ y = w. Since it has been established that x / y = z, the second crossing algebra can be altered to the form (x ∗ y) / y = w. Furthermore, arc w in the knot on the left side corresponds to the arc x on right side. Therefore, the final algebraic forms of the Type 2 Reidemeister Moves are then the following:

$$(X * Y)/Y = X \tag{2.3.1}$$

$$(X/Y) * Y = X \tag{2.3.2}$$

These equational forms correspond to the equational theorem of right cancellation.

2.4 Type 3 Reiedemeister Moves

The transformation in Figure 2.4.1 is the Type 3 Reidemeister Move. The left hand side of

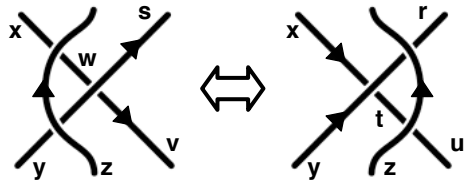


Figure 2.4.1. A Type 3 Reidemeister Move

Figure 2.4.1 contains 6 arcs, x,y,w,s,v and z. The right hand side contains the arcs x,y,t,z,u and r. On the left hand side, arc w is formed by z crossing over x to the left, yielding the crossing equation x ∗ z = w. Arc s results from arc z crossing over arc y to the left, forming the crossing equation y ∗ z = s. Arc v is formed by arc s crossing over arc w to the left, forming the crossing equation v = w ∗ s = (x ∗ z) ∗ (y ∗ z).

On the right hand side, arc t is formed by arc y crossing over arc x to the left, forming the crossing equation (x ∗ y) = t. Arc t then crosses over arc z to the left, resulting in u, yielding the crossing equation u = t ∗ z = (x ∗ y) ∗ z. Since the Reidemeister move says that v corresponds to u, leading to the following algebraic form:

$$(X * Y) * Z = (X * Z) * (Y * Z) \tag{2.4.1}$$

$$(X/Y)/Z = (X/Z)/(Y/Z) \tag{2.4.2}$$

2.5 Deriving New Equational Axioms from The Original Axioms

The following equational axioms represent the additional axioms that are derived from the three Reidemeister Moves.

1. X / (Y * Z) = ((X / Z) / Y) * Z
2. X * (Y / Z) = ((X * Z) * Y) / Z
3. X * (Y * Z) = ((X / Z) * Y) * Z
4. X / (Y / Z) = ((X * Z) / Y) / Z

The following proofs algebraically show the methods through which each of the delta rules are derived. All of the delta rules are built upon the original axioms that were derived from the Reidemeister Moves. These derivations were first performed in McGrail's paper on quandles [4].

$$\begin{aligned} X/(Y * Z) &= (X/Z) * Z)/(Y * Z) \\ &= ((((X/Z)/Y) * Y) * Z)/(Y * Z) \\ &= ((((X/Z)/Y) * Z) * (Y * Z))/(Y * Z) \\ &= ((X/Z)/Y) * Z \end{aligned} \tag{2.5.1}$$

$$\begin{aligned} X * (Y/Z) &= ((X * (Y/Z)) * Z) * (Y * Z) \\ &= ((X * Z) * (Y/Z) * Z)/Z \\ &= ((X * Z) * Y)/Z \end{aligned} \tag{2.5.2}$$

$$\begin{aligned} X * (Y * Z) &= ((X/Z) * Z) * (Y * Z) \\ &= ((X/Z) * Y)/Z \end{aligned} \tag{2.5.3}$$

$$\begin{aligned} X/(Y/Z) &= ((X/Z) * Z)/(Y/Z) \\ &= ((((X * Z)/Y) * Y)/Z)/(Y/Z) \\ &= ((((((X * Z)/Y)/Z) * Z) * Y)/Z)/(Y/Z) \\ &= ((((X * Z)/Y)/Z) * (Y/Z))/(Y/Z) \\ &= ((X * Z)/Y)/Z \end{aligned} \tag{2.5.4}$$

2.6 The Quandle Term Rewriting System

A quandle is a set of elements subject to the following equations: idempotence, right cancellation, and right self-distributivity [5]. It has now been seen that the Reidemeister moves are a way of transforming knots to see if they are ambient isotopic. From these Reidemeister moves there have been derived equational axioms that make up the theory of the quandle. Therefore, it is possible to see these equational axioms as way of transforming, or *rewriting* terms, in the same manner that Reidemeister moves transform the structure of a knot. Since Reidemeister moves can be used to check if one knot is the "deformation" of another, the equational axioms in the quandle can be used if one term can be made identical to the other. Thus the basis of the quandle *term rewriting system* will be to

use the equational axioms derived from the Reidemeister move to see if one term can be rewritten to another. The quandle term rewriting system will then be made up of the following equational axioms:

1. X * X = X
2. X / X = X
3. (X * Y) / Y = X
4. (X / Y) * Y = X
5. X / (Y * Z) = ((X * Z) / Y) / Z
6. X * (Y / Z) = ((X * Z) * Y) / Z
7. X * (Y * Z) = ((X / Z) * Y) * Z
8. X / (Y / Z) = ((X * Z) / Y) / Z

3

Syntactic Unification

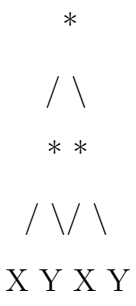
This chapter will discuss the properties of syntactic unification and discuss the concept of a syntactic unifier. Syntactic unification is method of unifying two terms by attempting to make them syntactically equivalent [13]. Two terms are considered "unified" when either term is of the exact same form of the other term. This means at every variable and non-variable position, as well as the respective operators of each term, the terms are exactly alike. A syntactic unifier is considered to be any collection of *substitutions* that make the two terms syntactically equivalent. More than one given syntactic unifier can exist for a given set of terms that are being unified. Substitutions can only occur at variable subterms within the term.

3.1 Terms, Subterms, and Roots

A term can be defined as some number of variables *rooted* at some operator. The phrase rooted refers to the structure of the abstract syntax tree of a given term. The "root" of the tree is the operator that is applied at the topmost level to it's arguments[11]. In other words, it is the operator that exists in the "middle" of the term: the location from which further subterms of the overall term can be parsed. For a simple example

of what a "root operator" looks like, consider the term X * Y. The root operator in this case is the * operator. It's left and right children are the X and Y values, the terms the operator operator is applied. The syntax tree, then, for the X * Y example, looks like this:

Consider an example of a syntax tree where the term is more complicated: A term where there are further terms rooted at the root operator that contain their own root operator. Terms the root operator are applied to are referred to as the *subterms*. Consider the term (X * Y) * (X * Y): the root operator is the *, but how can the syntax tree be recursively constructed from the root operator in this case? The syntax tree for this entire term is then:



It can be seen here then the root operator remains the * symbol, and the arguments the root operator is applied to are (X * Y) and (X * Y). The left and right children of the root operator are recursively defined as the respective syntax trees for the terms the root operator are applied to. These left and right children are the subterms of the entire term. Now it be can understood with more clarity what is meant by the references to the "root operator" and the "subterms".

3.2 Constraints on Syntactic Unification and the Occurs Check

For two terms to be syntactically unifiable, they need not be of the same length. At first it may be believed, intuitively, that two terms of varying lengths cannot be unified. This is since any term that contains more variable positions than the other cannot be made equivalent to the other term at these positions, and vice versa. However, any variable can be substituted not just for any other variable, but also,any other term. Consider a case where there exists a syntactic unifier for the two terms of differing length, X $*$ Y and Z

It is possible to substitute Z to become the term X * Y. Performing the substitution {Z $\rightarrow$ X * Y} causes Z to be transformed into the term X * Y, making this new term with the substitution applied equivalent to the other term. The substitution {Z $\rightarrow$ X * Y} is known as a *syntactic unifier*. It is a substitution, that when applied to both terms, leads to two terms that are syntactically identical. However, although there are large degrees of freedom in what are considered legitimate substitutions (as it can be seen, it possible to substitute singular variables for entire terms), there are cases in which two terms simply are not syntactically unifiable.

One way two terms can be not syntactically unified is if there exists some kind of endless looping caused by substitutions that attempt to make one term identical to another. The following is a simple example of an attempted syntactic unification sequence that will simply lead to endless looping in the domain of the syntactic unifier that is being derived. Consider the two terms: (Y, Y) and(X,f(X)).The strategy used in this example, and in all examples of syntactic unification is that of *unifying* each position in each term. Consider the first position in each term. The obvious first step is to send Y to X, and apply the substitution {Y $\rightarrow$ X}. This changes our term (Y,Y) into the term (X,X). Now, there exists a question of unifying the term (X,X) with (X,f(X)). The attempt to find a unifier for these two terms will show the basis for the occurs-check endless looping that will make them not unifiable. It would seem the only term left to make a substitution for would be to match X with f(X) by performing the substitution {X $\rightarrow$ f(X)}. It must be

remembered that substitutions must be applied to both and not only one of the terms- so the substitution is applied to both the (X,X) term and the (X,f(x)) term.

In an attempt to substitute f(X) for X, and endless loop of the following form occurs:

$$f(X) \rightarrow f(f(X)) \rightarrow f(f(f(X)))) \ ...$$

What is occurring ia an attempt to substitute a variable for a compound term that contains that original variable. Inevitably, this leads to an endless loop of generation of new compound terms. The compound term will create increasingly large compound terms that recursively contain the previous compound term. Thus, when doing syntactic unification, something that must always be avoided is avoiding substituting variables for compound terms when said compound term contains said variable. Avoiding such substitutions that lead to this pattern is known as performing the *occurs check*. [15].

Another bound on syntactic unification is the existence of generators. Generators are constants for which no substitutions can be made. This can lead to issues with unification with terms that contain less unique variable positions than there are unique generator positions in the other term. Generators are denoted by lowercase letters while variables are denoted by uppercase letters. A simple example of two terms that are not syntactically unifiable because of a lack of unique variable positions relative to unique generator positions are the terms (X,X) and (a,b). It possible to match X to a, or be able to match X to b, but it is simply impossible to match X with both a and b. And since there is only the variable X to match with two unique generators, there is no way to derive a syntactic unifier for these two terms.

4

Equational Unification

Syntactic unification covers the simple idea of attempting to unify the syntax of two different terms by applying some set of substitutions to each term. The concept of equational unification incorporates the usage of equational theorems to reach the end goal of syntactic unification. The transformation of terms now occurs in two stages: The narrowing stage and the rewriting stage. To derive an equational unifier for any two terms, there must be one or more narrowing and rewriting steps to achieve this goal (unless the two given terms are already exactly identical). Equational unification is the process of finding some substitution that makes two terms equivalent modulo some equational theory [10].

4.1 Narrowing and Rewriting

Narrowing is the stage of an equational unification step in which the structure of a given subterm becomes equivalent to the left hand side of a rewrite rule [10]. The redex of a rewrite rule is the left-hand side of a rewrite rule. A rewrite rule is any equational rule that denotes the transformation of a term structure. The rewrites rules of the quandle term rewriting system (TRS) are considered to be the axioms derived from the knot quandle, as

discussed in chapter 2. A narrowing step attempts to find a substitution for a given term or subterm that transforms it's structure into that of one of the redexes of the rewrite rule. Once the subterm's structure becomes equivalent to that of one of the redexes, a *rewrite* step can be performed.

The rewrite step is simply the application of the equational rule- the subterm previously narrowed to one of the redexes will be *rewritten* to match the structure of the righthand side (known as the *reduct*) of the rewrite rule that was chosen in the narrowing step [13]. The subterm that was transformed is notated by t|s, where s represents the position of the subterm within the term t. The process can be formally denoted with the following steps:

Given some TRS with a rewrite rule that has redex l and reduct r

Given some term T with a subterm t|s

1. Find some unifier, θ, that transforms the structure of t|s into that of some rewrite rule in A, $l \rightarrow r$

2. Apply the unifier θ to the subterm t|s.

3. Transform the structure of $(t|s)\theta$ into the respective reduct for the redex whose structure $(t|s)\theta$ is equivalent to

4. The new structure of (t|s) is now $(t|r)\theta$

4.2 Equational Unification Example

The following is a practical application of the narrowing/rewriting steps. The following two terms will be unified, in which A,B,C and D are all variables:

$$t_1 = (((A * B) * C) \ / \ D)$$

$$t_2 = A$$

Narrowing will first occur at the left subterm of the left subterm, $A * B$. This subterm's structure can be made equal to the structure of the idempotence redex, $X * X = X$. The following substitution will be applied:

$$[B \rightarrow A]$$

transforming the term into the new structure:

$$t_1 = (((A * A) * C) \ / \ D)$$

creating the new subterm A ∗ A, as this is equivalent to the structure of the idempotence redex rooted at the star operator. This subterm can now be rewritten to the idempotence reduct, creating the new term:

$$t_1 = ((A * C) \ / \ D)$$

This step was the rewriting step that followed the narrowing step at the subterm. First a substitution was applied making the structure of the subterm equivalent to the idempotence redex, and then the new subterm was rewritten to the respective idempotence reduct. Consider now the next narrowing and rewriting steps. Now the entirety of t_1 can be made equivalent to the structure of the redex of the right cancellation rule,

$$(X * Y) \ / \ Y$$

since there is a left subterm rooted at the ∗ operator and a right subterm operated at the / operator. By performing the substitution

$$[D \rightarrow C]$$

the term's structure will be come equivalent with the structure of the right cancellation rule, and the rewrite step will transform the narrowed term to the reduct of right cancellation rule redex the term shares an identical structure with, which is just t_2, A. This narrowing and rewriting step takes the form of the following transformations:

```
Narrowing Step:
(A * C) / C
Rewriting Step:
(A * C) / C rewrites to A
```

An equational unifier will then have been found for the two terms, the unifier:

θ= [B$\rightarrow$ A, D $\rightarrow$ C]

5

Rewrite Systems Properties

A rewrite system can also be proved to have the properties of *termination* and *confluence*. In this section, it will be proven that the quandle TRS has these two properties, and, furthermore, the properties of *convergence*. Please see Bader and Nipkow's book for further insight on term rewriting systems [14].

5.1 Terminaton Modulo Term Rewriting Systems

For a term rewrite system to be terminating, there must exist a *reduction order* such that, for all rules in the rewrite system, each redex is considered to be "larger" than it's respective reduct. It must be defined what it means for one term to be "larger" than another to prove that the rewrite system has a consistent reduction order throughout all of it's rules. The reduction order will be based upon the following recursive algebraic equation [4], and the size of a term is defined as it's "measure", or as m(t), where t is some term:

$$m(t) = 0$$

$$m(t) = (1 + m(left(t)) + 3(m(right(t))))$$

Left and right, in the definition of the equation, refer to the left and right subterms of the term. Consider a simple application of this measure equation in relation to the idempotence rules to prove that reduction ordering is maintained. The idempotence rule is:

$$X * X = X$$

Now, by the rules defined by the measure equation, the reduct of the idempotence rule would fall into the "base case" of the recursive algebraic equation as it is a singular variable with no left or right subterm. Therefore, the measure value of the reduct would be 0. Now, consider the redex, which falls into the non-base case definition as it is rooted at an operator and therefore contains a left and right subterm:

$$m(t)=(1+m(X)+3m(X))$$
$$m(t)=(1+0+3(0)$$
$$m(t)=(1+0+0)$$
$$m(t)=1$$

Therefore, an ordering is maintained where the redex has a measure value of 1 and the reduct has a measure value of 0. This intuition seems obvious for both of the rules (idempotence and right self-cancellation) in which the redex is some term rooted at an operator and the reduct is a singular variable- of course a the term with more variables would be considered "larger". However, what about the case of the delta rules, in which the number of variables in the reduct are larger than the number of the variables in the redex? The measure equation weighs far more heavily (by a factor of 3) the size of the right subterm of a term. Therefore, so long as the reduct grows the amount of variables in it's left subterm up to a size relative to 3 times the right subterm of it's respective redex, it's still considered smaller. As an example, compare the measures of the redex and reduct of the rewrite rule:

$$X*(Y*Z)=(((X/Z)*Y))/Z$$

$$m(X*(Y*Z)=(1+m(X))+3m(Y*Z)$$
$$m(X*(Y*Z)=(1+0+3(1+m(Y)+m(Z)))$$
$$m(X*(Y*Z)=(1+0+3(1+0+0))$$
$$m(X*(Y*Z)=4$$

$$m(((X/Z)*Y))/Z=(1+m((((X/Z)*Y)))+3m(Z)$$
$$m(((X/Z)*Y))/Z=(1+(1+m(X/Z)+3m(Y))+3(0))$$
$$m(((X/Z)*Y))/Z=(1+(1+(1+m(X)+3m(Z)))+0)$$
$$m(((X/Z)*Y))/Z=(1+(1+(1+0+3(0)))+0)$$
$$m(((X/Z)*Y))/Z=3$$

Once again, the measure function proves for the case in which the reduct has more variables than it's redex that reduction ordering is maintained. The ordering is strict and is maintained throughout each of the rewrite rules. The weighting of the right subterm means that any rewrite rule must attempt to either outright reduce the number of variables in the rewriting of it's redex to reduct, or shift many of these variables to it's left subterm in the reduct. Furthermore, the existence of a lower bound on the measure of a term (a value of 0 for a singular variable or constant) proves that any rewrite sequence can eventually reach a reduced form, and over the course of any rewriting sequence, have a measure value that is monotonically non-increasing [6].

5.2 Confluence and Critical Pairs

It has now been proved the term rewriting system is strongly normalized by way of the measure function which shows that the reduction order ensures that for any rewrite sequence, the measure of each new term will not be any larger than the previous term. Newmann's Lemma states that for any strongly normalized terminating rewriting system,

proving that all of the system's *critical pairs* are locally confluent proves that the system is convergent. By Newmann's Lemma, so long as the rewriting system is terminating, proving local confluence over all critical pairs is enough to prove non-localized confluence over the entire rewriting system [14].

5.3 Confluence

The question of narrowing the subterms of a redex is inherently non-deterministic, since any term may contain subterms that can match to various redexes that exist in the rewriting system. There will exist terms that contain subterms that match to redexes that will "*interfere*" with each other. Two redexes interfere with each other if the transformation of one redex to it's respective reduct perturbs the structure of the other redex [14]. If no such interference exists, it means that there exist multiple paths to a unique normal form for the term. Consider a rewriting system with the rules:

$$f(x) \rightarrow f'(x)$$
$$g(x) \rightarrow g'(x)$$

Given the term f(g(x)), applying the second rewrite rule would transform g(x) to g'(x), but this would not prohibit us from applying the first rewrite rule to alter the topmost functor, f(x). Therefore, it is possible to apply first either the first rewrite rule to achieve the term f'(g(x)), or apply the second rewrite rule to achieve the term f(g'(x)), but no matter what rule is applied first, since the two redexes do not interfere with each other, the same normal form is always reached: f'(g'(x)).

It is easy to see that, no matter which sequence occurs in the rewriting process, the same normal form is always reached. This is the basis of the theorem of confluence: that although there exists multiple possible rewriting sequences, the same normal form will always be reached. This relation can be seen in the structure of a directed graph, in which

the nodes are terms, and the edges are rewrite steps. In the Figure 5.3.1, A rewrites to

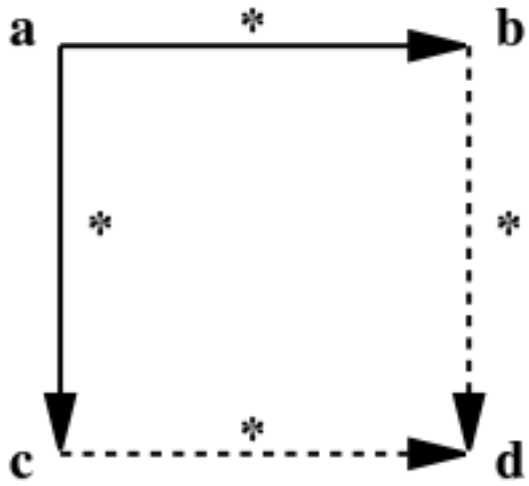


Figure 5.3.1. Directed Graph Showing Confluence Relation

both B and C, but is confluent since both B and C rewrite to D.

5.4 Newmann's Lemma

In the previous section it was seen through the example of the small f(x) and g(x) TRS that some term rewriting systems contain the axiom of confluence. Convergence is the combination of the axioms of termination and confluence. Convergence indicates any rewriting sequence modulo the rewriting system is guaranteed to both eventually no longer able to be reduced after an arbitrary number of rewrites, and that it is guaranteed to reach the same normal form regardless of of the order of rewrite steps it took (confluence). Newmann's Lemma states that if a rewriting system is terminating, it is required only to prove that the critical pairs are locally confluent to prove convergence [14]. But what is a critical pair? A critical pair is some term whose structure allows for it to be narrowed to two different redexes in a TRS. Consider two substitutions, $\sigma 1$ and $\sigma 2$, and two redexes of a rewrite system, l1 and l2, and a non-variable position in l1, p, denoted by l1 |p. The structure of a term that has a critical pair is then [14]:

$$\sigma 1(\mathrm{l1}|\mathrm{p})=\sigma 2\mathrm{l2}$$

Critical pairs are caused by *critical overlaps* [14]. Critical overlaps occur when narrowing a term to one redex perturbs the structure of the other redex it could be narrowed to. An example of this case in the quandle TRS would be the term:

$$(X * Y) * (X * Y)$$

Although intuitively it may seem that this is an obvious case of the idempotence redex, it is possible as well to invoke the delta rule redex, in which the overall term and right subterm are rooted at the $*$ operator.

Proving the local confluence of critical pairs then means proving that, regardless of the narrowing step taken, the same normal form is still reached. It was proved in the McGrail paper that all critical pairs in the quandle TRS were confluent [4], thus proving the validity of Newmann's Lemma for the quandle TRS. The existence of the axiom thus has the following implications for the rewrite system:

1. Every rewriting sequence, regardless of the order of narrowing and rewriting steps, will always reach the same normal form

2. For every rewrite sequence, the reduction order of the rewrite rules ensures that there exists a normalized form whose measure value can not further be reduced

6

The Maude System and The Finite Variant Property

One specialized tool for working with rewriting systems is the Maude programming language. This chapter includes an implementation of the quandle TRS in Maude.

6.1 Syntax Basics: Sorts and Subsorts

Maude itself is not statically typed and types in Maude are referred to as "sorts". Maude contains pre-defined importable numerical sort modules (as well as boolean modules) that provide mathematical operations. Maude also contains a system of "subsorting", a way of pseudo-encapsulating sorts. Consider the example of values of sort "NAT" representing Natural Numbers, and "NAT3", representing all natural numbers that are multiples of 3. The values of NAT3 are of course a subset of the values of NAT, since the natural numbers that are a multiple of 3 are a subset of all natural numbers in existence. Therefore, the NAT3 sort is a subsort of the NAT sort. Furthermore, all declarations in Maude must be followed by a spaced period. The following example code snippet shows how these sorts and subsorts are declared.

```
sort NAT .
sort NAT3 .
```

```
subsort NAT3 < NAT .
```

6.2 Syntax Basics: Operators, Variables, and Constants

Maude allows for the declarations of variables, constants, and user-defined operators. Both variable and constant declaration require the user to specify what sort the variable or constant will inhabit. Variable declaration requires the user to use the keyword "var", followed by the identifier they'll be using, and then a colon, followed by the identifier's sort. An example declaration of a variable would be the following:

```
var A : NAT .
```

Operator declaration in Maude is flexible, allowing for infix, prefix, and postfix operator declaration. Operator declaration in Maude is denoted by the "op" keyword, followed by the identifier for the operator. The infix/posfix/prefix nature of the operator is defined by where the arguments to the operator will exist by placing a variable number of underscores in relation to the operator. Underscores represent where the arguments will exist. After declaring the prefix/infix/postfix nature of the operator, the sort of the arguments will follow, as well as the sort the operator will return. The following code example shows how a prefix, postfix, and infix operator could be declared, all of which take in 2 arguments, and return one value.

```
op _*_ : NAT NAT -> NAT3 .
op == _ _ : NAT NAT -> BOOL .
op _ _ + : NAT NAT -> NAT .
```

The first defined operator here represents an infix * operator that takes in two arguments of sort NAT and returns a singular NAT3 value. The second operator represents a prefix equality operator that takes in the arguments of sort NAT and returns a boolean value. The last operator is defined as postfix addition, taking in two arguments of sort NAT and returning a value of sort NAT as well. Declaring a constant also requires similar syntax usage

as declaring operators. Constant declaration uses the same syntax as operator declaration, but omits any underscores representing argument placement, as well as omitting input types. The return type remains, as it now represents the sort of the constant that has been declared. Consider the following example of defining a constant "Zero", of sort NAT:

```
op Zero : -> NAT .
```

Variable, constant, and operators combine to form the basis of the formation of defining further, more complex expressions in Maude.

6.3 Functional Modules and Equations

Modules are Maude's way of abstracting data. Maude has two main types of modules: functional modules and system modules. Functional modules define data types and operators (the sort/subsort system discussed in the previous section), as well as *equations* and *membership axioms*. Functional modules are declared by the keyword "FMOD" followed by the module identifier and then the keyword "is". This denotes the beginning of a functional module. The module is terminated by the keyword "endfm". Membership axioms are a form of sort constraint: they specify a condition or conditions a term can follow that denotes the sort it belongs to.

Membership axioms are denoted by the keyword "mb" followed by some constraint, and then the ":" symbol, and finally, the sort the constraint defines. Maude also uses equations which are denoted by the "eq" keyword. These equations have a structure similar to the rewrite rules defined in the quandle TRS. Like all rewrite rules, they must have some redex that rewrites to a reduct, so there must exist a left-hand and right-hand side for every equation. The redex of an equation and it's respective reduct are separated by the "=" symbol. Often times, equations specify the behavior of an already declared operator. Operator declarations require a specification of a sort for the arguments that it takes

in and the sort for the argument it outputs. The following code snippet demonstrates a functional module that contains some operator and equation definitions:

```
fmod TEST_AXIOMS is
        sort Int .
        vars X Y Z : Int .
        op _*_ : Int Int -> Int .
        op _/_ : Int Int -> Int .
        op eq _ _ : Int Int -> Int .
        eq : X * X = X .
        eq : X / X = X .
endfm
```

It again should be noted there was no specification on how such operators actually evaluated their arguments. This is where equations come in. It is possible to declare equations in the field of pure rewriting logic: as seen in the example code snippet below, they are used to denote the knot quandle's rewrite rules. Equations can share an identifier with an already declared operator and then specify the behavior of said operator. Consider the following code snippet, which defines a functional module that encapsulates the rewrite rules of the quandle TRS, encoded as equations.

```
fmod SPROJ is
        protecting INT .
        sort Term .
        sort TermList .
        subsort Term < TermList .
        var A : Term .
        var B : Term .
        op _*_ : Term Term -> TermList .
```

```
        op _/_ : Term Term -> TermList .
        eq X * X = X  [variant] .
        eq X / X = X  [variant] .
        eq ( X * Y ) / Y = X  [variant] .
        eq ( X / Y ) * Y = X  [variant] .
        eq X / ( Y / Z ) = ( ( X * Z ) / Y ) / Z  [variant] .
        eq X * ( Y / Z ) = ( ( X *  Z ) * Y ) / Z  [variant] .
        eq X * ( Y * Z ) = ( ( X / Z ) * Y ) * Z  [variant] .
        eq X / ( Y * Z ) = ( ( X / Z ) / Y ) * Z  [variant] .
        eq X / ( Y / Z ) = ( ( X * Z ) / Y ) / Z  [variant] .
endfm
```

The Term and TermList sorts will come into play more in the discussion of the system module, but in a rudimentary sense can be thought of in pure parsing terms as such: Terms represent singular atoms, such as any variable or constant, whilst TermLists refer to any term rooted at one of the binary operators in the quandle which of course are $*$ and $/$. A singular TermList in the context of the functional module would be "X $*$ Y", where X and Y are both Terms. To achieve successful parsing, the program uses the "mb" keyword, which signifies sort membership. Usage of this keyword denotes that the argument passed to they keyword's structure indicates the argument is of a given sort. Consider the following code snippet, which defines the membership rules for Terms and TermLists:

```
mb (X:Term * Y:Term) : TermList .
mb (X:Term / Y:Term) : TermList .
mb (X:TermList * Y:Term) : TermList .
mb (X:TermList / Y:Term) : TermList .
mb (X:Term * Y:TermList) : TermList .
mb (X:Term / Y:TermList) : TermList .
```

```
mb (X:TermList * Y:TermList) : TermList .
mb (X:TermList / Y:TermList) : TermList .
```

The quandle TRS module must have the ability to parse the left and right subterms of any given term. The following code snippet exemplifies the operator and equation declarations that give the program this ability:

```
op lhs _ : Term -> Term .
op rhs _ : TermList -> Term .
eq lhs(X:TermList * Y:TermList) = X:TermList .
eq lhs(X:TermList / Y:Term) = X:TermList .
eq rhs(X:Term * Y:Term) = Y:Term .
eq rhs(X:Term / Y:Term) = Y:Term .
eq rhs(A) = A .
eq lhs(B) = B .
```

The lhs and rhs equations return the left hand subterm and righthand subterm of any TermList. Of course, in the simple case, like the X $*$ Y TermList above, it would return a singular Term. There could exist many TermLists that contain an identical structure to the term ((a $*$ (b $*$ c) / X)), where the left hand subterm is a TermList, while the right-hand side is a singular Term. Therefore, there are multiple re-definitions of the lhs and rhs equations. This shows that Maude also allows for pattern matching through it's flexibility of equational re-definitions. Rather than writing a singular equation with multiple conditions, there can be multiple equational definitions of one operator. The functional module also contains what seems like a re-definition of variables never explicitly declared. The variables X, Y, and Z appear many times in the equations for the rewrite rules and the membership axioms. Consider the following code snippet that exemplifies this phenomeon:

```
mb (X:Term * Y:Term) : TermList .
```

```
mb (X:Term / Y:Term) : TermList .
mb (X:TermList * Y:Term) : TermList .
mb (X:TermList / Y:Term) : TermList .
mb (X:Term * Y:TermList) : TermList .
mb (X:Term / Y:TermList) : TermList .
mb (X:TermList * Y:TermList) : TermList .
mb (X:TermList / Y:TermList) : TermList .
eq lhs(X:TermList * Y:TermList) = X:TermList .
eq lhs(X:TermList / Y:Term) = X:TermList .
eq rhs(X:Term * Y:Term) = Y:Term .
eq rhs(X:Term / Y:Term) = Y:Term .
```

This shows an example of the scoping of variables in the Maude system. The variables defined through the syntax "Identifier:Sort" represent a local definition of a variable only to that equational definition or membership axiom. Maude is able to parse these locally defined variables without issue even if they share the same name as the globally defined variables as they only have a scope of their current equation. The protecting keyword at the beginning of the module also signifies the importation of the numerical integer module from C++ that allows for execution of mathematical operations. The final set of declarations in the functional module are the rewrite rules of the quandle TRS. They are followed by the special "variant" keyword- which will be explained in depth in the following sections. Finally, here is the full functional module for the quandle TRS:

```
fmod SPROJ is
        protecting INT .
        sort Term .
        sort TermList .
        subsort Term < TermList .
        var A : Term .
```

```
vars X Y Z : Term .

var B : Term .

op _*_ : Term Term -> TermList .

op _/_ : Term Term -> TermList .

op lhs _ : Term -> Term .

op rhs _ : TermList -> Term .

mb (X:Term * Y:Term) : TermList .

mb (X:Term / Y:Term) : TermList .

mb (X:TermList * Y:Term) : TermList .

mb (X:TermList / Y:Term) : TermList .

mb (X:Term * Y:TermList) : TermList .

mb (X:Term / Y:TermList) : TermList .

mb (X:TermList * Y:TermList) : TermList .

mb (X:TermList / Y:TermList) : TermList .

eq lhs(X:TermList * Y:TermList) = X:TermList .

eq lhs(X:TermList / Y:Term) = X:TermList .

eq rhs(X:Term * Y:Term) = Y:Term .

eq rhs(X:Term / Y:Term) = Y:Term .

eq rhs(A) = A .

eq lhs(B) = B .

eq X * X = X  [variant] .

eq X / X = X  [variant] .

eq ( X * Y ) / Y = X  [variant] .

eq ( X / Y ) * Y = X  [variant] .

eq X / ( Y / Z ) = ( ( X * Z ) / Y ) / Z  [variant] .

eq X * ( Y / Z ) = ( ( X *  Z ) * Y ) / Z  [variant] .

eq X * ( Y * Z ) = ( ( X / Z ) * Y ) * Z  [variant] .
```

```
        eq X / ( Y * Z ) = ( ( X / Z ) / Y ) * Z  [variant] .
        eq X / ( Y / Z ) = ( ( X * Z ) / Y ) / Z  [variant] .
endfm
```

6.4 System Modules

System modules contain all the functionality of function modules, but introduce the concept of "rules". To declare a system module, there must be the keyword "MOD" followed by the module identifier, followed by the keyword "is", much like the functional module. It is terminated by the keyword "endm". Of course, it is preferable for the system module to have access to the declarations, sorts, and operators in the functional module. To perform this importation, simply use the keyword "INCLUDING" followed by the functional module's identifier. For the sake of this program, the quandle's rewrite rules are declared as equations so they could access the "variant" attribute, since this program will be attempting to use a variant-based narrowing approach (which will be explained more in-depth in the coming sections).

Maude does not allow for the usage of the variant attribute for rules, constraining the declaration of the rewrite rules as equations rather than rules. Furthermore, the system module also allows for the usage of the frozen attribute: which ensures that Maude will not attempt to rewrite any of the given arguments in an operator. This attribute came into play the system module defines operator/equation pairs to evaluate the "measure", discussed in chapter 5, of a given term, and another operator to compare the measure of two given terms and return if one is greater than the other. The measure equation is defined recursively: The first and most simple case (the base case in a sense) is that of the measure of a singular variable or constant. As specified in the measure equation, this simply returns 0. The second case is that of a simple TermList, in which there exist two Terms (singular variables or constants) rooted at one of the binary operators. This is defined as being 1 + measure(left subterm) + 3 * measure(right subterm). The remaining

cases are all defined in the same manner, with cases ranging from both subterms being TermLists, to only one of them. The following code snippet exemplifies the declaration of the measure equation:

```
eq m(X:Term) = 0 .
eq m(x:Term) = 0 .
eq m(X:Term * Y:Term) = 1 + m(lhs(X:Term * Y:Term)) + 3 *
 ↪  m(rhs(X:Term * Y:Term)) .
eq m(X:Term / Y:Term) = 1 + m(lhs(X:Term * Y:Term)) + 3 *
 ↪  m(rhs(X:Term * Y:Term)) .
eq m(X:TermList * Y:Term) = 1 + m(X:TermList) + 3 * m(Y:Term) .
eq m(X:TermList / Y:Term) = 1 + m(X:TermList) + 3 * m(Y:Term) .
eq m(X:TermList * Y:TermList) = 1 + m(X:TermList) + 3 *
 ↪  m(Y:TermList) .
eq m(X:TermList / Y:TermList) = 1 + m(X:TermList) + 3 *
 ↪  m(Y:TermList) .
```

A singular term of course reduces to simply 0, while any term rooted at a binary operator is recursively defined as 1 + the measure of the left hand side + 3 times the measure of the right hand side. As an example of this equation in action, consider the measure of a Term (defined as the sort TermList in the code, a Term refers to a singular variable/constant). To actually get the equation to evaluate, the "reduce" command must be invoked which reduces the statement it is given:

```
Maude> reduce m ((X * A) * Z) .
reduce in PRUNE : m ((X * A) * Z) .
rewrites: 14 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 2

Maude> reduce m (X * Y) .
```

```
reduce in PRUNE : m (X * Y) .
rewrites: 8 in 0ms cpu (0ms real) (~ rewrites/second)
result NzNat: 1

Maude> reduce m (X) .
reduce in PRUNE : m X .
rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
result Zero: 0
```

The above terminal output illustrates the usage of the m (measure) equation. The equation correctly evaluates the measure of a given term. The next built in operator/equation is that of comparing the measure of two terms. For this, Maude's conditional abilities must be utilized. Maude allows for equations to have return values based on some "if then else" statement. Such a statement is evaluated in the structure "if (condition) then true/false (Maude has a built in Boolean 'Bool' sort) else true/false fi". The fi keyword must be included to denote the end of the else statement. The structure of the comparison equation, denoted as "cmp", would return true if the measure of the first given formal parameter was larger than the measure of the second formal parameter. The following code snippet shows how these operators were declared:

```
op cmp _ _ : Term Term -> Bool [frozen] .
op cmp _ _ : TermList Term -> Bool [frozen] .
op cmp _ _ : TermList TermList -> Bool [frozen] .
op cmp _ _ : Term TermList -> Bool [frozen] .
eq cmp X:Term Y:Term = if m(X:Term) > m(Y:Term) then true else
↪  false fi .
eq cmp X:TermList Y:Term = if m(X:TermList) > m(Y:Term) then true
↪  else false fi .
```

```
eq cmp X:Term Y:TermList = if m(X:Term) > m(Y:TermList) then true
↪  else false fi .
eq cmp X:TermList Y:TermList = if m(X:TermList) > m(Y:TermList)
↪  then true else
false fi .
```

There existed 4 different cases, and therefore overloadings of the operator, to account for. These were the cases of comparing two Terms, the case of the first argument being a Term and the second being a TermList, the case of the first argument being a TermList and the second being a Term, and the final case, comparing two TermLists. The following terminal output shows an example example of the comparison operator/equation

```
Maude> reduce cmp (X * Y) X .
reduce in PRUNE : cmp X * Y X .
rewrites: 12 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true

Maude> reduce cmp X (X * Y).
reduce in PRUNE : cmp X (X * Y) .
rewrites: 12 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: false

Maude> reduce cmp ((X * Y) * (Y * Z)) (Z * X) .
reduce in PRUNE : cmp (X * Y) * (Y * Z) (Z * X) .
rewrites: 32 in 0ms cpu (0ms real) (\~{} rewrites/second)
result Bool: true
```

The entire system module implemetation is as follows:

```
mod PRUNE is
        including SPROJ .
        op m _ : Term -> Int [frozen] .
        op m _ : TermList -> Int [frozen] .
        vars X Y Z : Term .
        eq m(X:Term) = 0 .
        eq m(x:Term) = 0 .
        eq m(X:Term * Y:Term) = 1 + m(lhs(X:Term * Y:Term)) + 3 *
        ↪  m(rhs(X:Term * Y:Term)) .
        eq m(X:Term / Y:Term) = 1 + m(lhs(X:Term * Y:Term)) + 3 *
        ↪  m(rhs(X:Term * Y:Term)) .
        eq m(X:TermList * Y:Term) = 1 + m(X:TermList) + 3 * m(Y:Term) .
        eq m(X:TermList / Y:Term) = 1 + m(X:TermList) + 3 * m(Y:Term) .
        eq m(X:TermList * Y:TermList) = 1 + m(X:TermList) + 3 *
        ↪  m(Y:TermList) .
        eq m(X:TermList / Y:TermList) = 1 + m(X:TermList) + 3 *
        ↪  m(Y:TermList) .
        op cmp _ _ : Term Term -> Bool [frozen] .
        op cmp _ _ : TermList Term -> Bool [frozen] .
        op cmp _ _ : TermList TermList -> Bool [frozen] .
        op cmp _ _ : Term TermList -> Bool [frozen] .
        eq cmp X:Term Y:Term = if m(X:Term) > m(Y:Term) then true else
        ↪  false fi .
        eq cmp X:TermList Y:Term = if m(X:TermList) > m(Y:Term) then true
        ↪  else false fi .
        eq cmp X:Term Y:TermList = if m(X:Term) > m(Y:TermList) then true
        ↪  else false fi .
```

```
eq cmp X:TermList Y:TermList = if m(X:TermList) > m(Y:TermList)
↪  then true else false fi .
endm
```

6.5 Variants

For any given term, it is possible can generate an arbitrary number of variants, dependent on the structure of the quandle TRS. A variant is defined to be a pair in which the first element is the normalized form of the term, and the second item being the substitution that leads to the transformation of the term, through some applications of rewriting steps, to it's normalized form. The structure of a variant is:

$$(\text{normal form}, \theta)$$

One example of a variant for the term (X $*$ Y) would be the pair:

$$(\text{X},[\text{Y}\rightarrow\text{X}])$$

As substituting Y to X transforms the term into the redex of the idempotence rule, which rewrites to the normalized term X. Of course, it is possible to generate very many variants, possibly even an infinite amount. It would be of great interest to know if a given rewrite system does have the finite variant property. The existence (or lack thereof) the existence of the finite variant property is especially crucial within the context of Maude. The presence of the "variant" attribute in brackets that followed the definitions of the rewrite rules (or in the context of the functional module, equations) can now be fully explained.

```
eq X * X = X  [variant] .
eq X / X = X  [variant] .
eq ( X * Y ) / Y = X  [variant] .
eq ( X / Y ) * Y = X  [variant] .
eq X / ( Y / Z ) = ( ( X * Z ) / Y ) / Z  [variant] .
```

```
eq X * ( Y / Z ) = ( ( X *  Z ) * Y ) / Z  [variant] .
eq X * ( Y * Z ) = ( ( X / Z ) * Y ) * Z  [variant] .
eq X / ( Y * Z ) = ( ( X / Z ) / Y ) * Z  [variant] .
eq X / ( Y / Z ) = ( ( X * Z ) / Y ) / Z  [variant] .
```

The variant attribute signals to the Maude system that the given rewrite rule can be used in the variant generation process. The finite variant property itself is only a semi-decidable problem: However, there does exist a semi-decidable procedure to identify when a rewrite system does contain the finite variant property. The boundedness property states that for any given term, there must an upper bound on the length of a rewrite sequence to reach a normalized form. Consider the relation of the boundedness property to the term X $*$ Y:

```
X * Y -> X
Variant: (X, [X->Y])
Derivation Bound: 1 rewriting deriation (narrowing to the idempotence
↪  redex)
```

It may at first appear the quandle TRS respects the boundedness property. Yet, this is not the case. Such a lack of a property leads to infinite variant generation since it leads to infinite rewriting capabilities. This is because it allows for rewriting in such a manner endless narrowing steps can be taken at any term structure- there is no one normalized substitution which "bounds" the possibility of the length of a rewrite sequence. In fact, the knot quandle does not contain the finite variant property when this axiom is checked. It is the "delta rules", or, in other words, the non-idempotent or right cancellation rules that break the finite variant property. The following rules define the delta rules:

```
eq X / ( Y / Z ) = ( ( X * Z ) / Y ) / Z  [variant] .
eq X * ( Y / Z ) = ( ( X *  Z ) * Y ) / Z  [variant] .
eq X * ( Y * Z ) = ( ( X / Z ) * Y ) * Z  [variant] .
eq X / ( Y * Z ) = ( ( X / Z ) / Y ) * Z  [variant] .
```

Consider a simple example showing the lack of the boundedness property. The term (X $*$ Y) should take only one narrowing and rewriting step to reach a normalized form, X. Therefore, it has a variant (X, {Y$\rightarrow$X}). Now, consider the substitution $\{Y \rightarrow$T $*$ Z}). Such a substitution leads to the invocation of the delta rule of the structure (X $*$ (T $*$ Z)). The existence of the delta rules then, with a structure that allows for narrowing to always occur at the right subterm, always allows the quandle TRS to narrow in this manner. Through this "loophole", it is wholly possible to take endless narrowing steps that invoke the delta rules which will constantly grow the size of the term and prevent it from ever reaching a normalized form. Consider the following rewrite sequence that shows the problem the delta rules pose:

```
Original Term: X * Y
Narrowing Substitution: [Y-> T * Z]
X * (T * Z) ->  ( ( X / Z ) * T ) * Z
New Term: ( ( X / Z ) * T ) * Z
Narrowing Substitution Modulo Delta Rule:
[Z-> A * B]
New Term:((X / (A * B)) * T) * (A * B)
((X / (A * B)) * T) * (A * B) ->   ((((X / (A * B)) * T) / B) * A) * B
```

This narrowing and rewrite sequence is an endless narrowing sequence modulo the delta rules. In the first rewrite step, the original redex is narrowed to the the delta rule redex which rewrites to the respective reduct:

```
( ( X / Z ) * T ) * Z
```

At first glance, it may appear that the quandle TRS can now stop narrowing, since the previous term is a reduct of one of the original rewrite rule redexes. However,the quandle TRS will always be able to narrow to the delta rules regardless of the structure of any term because of the right associative nature of the delta rule redexes. It can be seen in the

previous rewrite derivations the right hand subterm of the topmost operator can simply be substituted out for the right hand side of a delta rule redex, and thus allow for an invocation of the delta rule. In simplified terms, delta rule redexes have this structure:

```
X BINARY_OPERATOR (A BINARY_OPERATOR B)
```

And every term in the context of the quandle TRS has the structure:

```
X BINARY_OPERATOR Y
```

Thus, [Y → A BINARY_OPERATOR B] will always be a possible narrowing step for the quandle TRS to take. This therefore breaks the boundedness property constraint: The TRS has no upper bound for the number of derivations to reach a normalized form.

6.6 Variant-Based Narrowing in Maude

Maude can generate the variants of any given term, through it's "get variants" command. To generate such variants, it is required to modify the functional module previously declared in this chapter. Declaring two differing sorts, Term and TermList, prevent the narrowing of a lower sort (Term) to a higher sort (TermList).This bounds the variant generation, but is not desired, as greatly limiting Maude's variant generation ability hinders it's unification abilities. In the previous section, it was noted that a legitimate narrowing and rewriting sequence is:

```
Original Term: X * Y
Normalized Substitution: [Y-> T * Z]
X * (T * Z) ->  ( ( X / Z ) * T ) * Z
New Term: ( ( X / Z ) * T ) * Z
```

It also should be noted that in the original functional module, Terms and TermLists' membership axioms were denoted like so:

```
sort Term .
```

```
sort TermList .
subsort Term < TermList .
mb (X:Term * Y:Term) : TermList .
mb (X:Term / Y:Term) : TermList .
mb (X:TermList * Y:Term) : TermList .
mb (X:TermList / Y:Term) : TermList .
mb (X:Term * Y:TermList) : TermList .
mb (X:Term / Y:TermList) : TermList .
mb (X:TermList * Y:TermList) : TermList .
mb (X:TermList / Y:TermList) : TermList .
```

Therefore, by the logic of the original functional module, Y, a Term, could not narrow to (T $*$ Z), a TermList, as Term is a subsort of TermList. This artificial constraint would incorrectly bound the variants search tree, so therefore to successfully generate all possible variants subsorting would be omitted to avoid over-pruning. The following file simply denotes one sort, Term, rather than two sorts of different levels of order-sorting.

```
fmod VARIANT_MODULE is
        sort Term .
        vars X Y Z : Term .
        op _*_ : Term Term -> Term .
        op _/_ : Term Term -> Term .
        eq X * X = X  [variant] .
        eq X / X = X  [variant] .
        eq ( X * Y ) / Y = X  [variant] .
        eq ( X / Y ) * Y = X  [variant] .
        eq X / ( Y / Z ) = ( ( X * Z ) / Y ) / Z  [variant] .
        eq X * ( Y / Z ) = ( ( X *  Z ) * Y ) / Z  [variant] .
        eq X * ( Y * Z ) = ( ( X / Z ) * Y ) * Z  [variant] .
```

```
        eq X / ( Y * Z ) = ( ( X / Z ) / Y ) * Z  [variant] .
        eq X / ( Y / Z ) = ( ( X * Z ) / Y ) / Z  [variant] .
endfm
```

Maude itself has built in unification capabilities- it's "variant unify" command. Said command uses *variant-based narrowing*. Variant-based narrowing considers that, for any two terms, if one term is exists in the variant space of another, the two must be unifiable. Consider that for any terms t and t', if t' can be reached in some rewrite sequence from t, then in t's variant space, there must exist the variant:

$$(t', \theta)$$

.

As a consequence of this narrowing strategy, Maude's variant unify command requires Maude to fully generate all variants before checking if the term which is being unified with exists in the variant space. This is a problem, as the TRS as currently constructed generates massive amounts of variants due to the lack of the boundedness property. Therefore, utilizing Maude's "get variants" command shows Maude will struggle to successfully generate all variants as the computational demand to so is massive. The "get variants" command shows the number of rewrites that were needed to reach a normalized form, as well as the actual normalized form it reached. It can be seen Maude will generate massively complex terms that require increasingly long rewrite derivations.

6.7 Maude Variant Generation and Unification

Consider the more complex variants Maude generates for the term (X $*$ Y). Maude can generate huge amounts of complex variants due to the delta rules. In fact, the get variants command would never terminate unless manually halted. As discussed in the description of variant-based narrowing, Maude needs to generate a full variant space before it can check if two terms are unifiable modulo the variant unify command. However, since the

```
Variant #963
rewrites: 2532 in 1076ms cpu (1080ms real) (2353 rewrites/second)
Term: ((((((((((((#1:Term * #3:Term) / #4:Term) / #5:Term) / #6:Term) /
    #7:Term) * #8:Term) * #9:Term) / #8:Term) * #7:Term) * #6:Term) * #5:Term)
    * #4:Term) / #3:Term) / #2:Term
X --> #1:Term / #2:Term
Y --> ((((((#9:Term / #8:Term) * #7:Term) * #6:Term) * #5:Term) * #4:Term) /
    #3:Term) / #2:Term

Variant #964
rewrites: 2532 in 1076ms cpu (1080ms real) (2353 rewrites/second)
Term: ((((((((((((((((#3:Term * #4:Term) * #5:Term) / #6:Term) / #7:Term) /
    #8:Term) / #9:Term) / #10:Term) * #2:Term) * #1:Term) / #2:Term) *
    #10:Term) * #9:Term) * #8:Term) * #7:Term) * #6:Term) / #5:Term) / #4:Term
X --> #3:Term
Y --> (((((((#1:Term / #2:Term) * #10:Term) * #9:Term) * #8:Term) * #7:Term) *
    #6:Term) / #5:Term) / #4:Term

Variant #965
rewrites: 2532 in 1076ms cpu (1080ms real) (2353 rewrites/second)
Term: ((((((((((((((((#3:Term * #4:Term) * #5:Term) / #6:Term) / #7:Term) /
    #8:Term) / #9:Term) / #10:Term) / #2:Term) * #1:Term) * #2:Term) *
    #10:Term) * #9:Term) * #8:Term) * #7:Term) * #6:Term) / #5:Term) / #4:Term
X --> #3:Term
Y --> (((((((#1:Term * #2:Term) * #10:Term) * #9:Term) * #8:Term) * #7:Term) *
    #6:Term) / #5:Term) / #4:Term

Variant #966
rewrites: 2532 in 1076ms cpu (1080ms real) (2353 rewrites/second)
Term: ((((((((((((#1:Term * #2:Term) / #3:Term) / #4:Term) / #5:Term) /
    #6:Term) / #7:Term) * #8:Term) * #7:Term) * #6:Term) * #5:Term) * #4:Term)
    * #3:Term) / #2:Term) / #1:Term
X --> #1:Term
Y --> ((((((#8:Term * #7:Term) * #6:Term) * #5:Term) * #4:Term) * #3:Term) /
    #2:Term) / #1:Term
```

Figure 6.6.1. Variants generated for the term X ∗ Y

variant generation process is non-terminating for the quandle TRS, the variant unify command is of no use. Due to the quandle TRS' lack of the boundedness property, as well as Maude's generate-all-variants first method of performing variant-based narrowing, Maude's variant-based unification abilities will not succeed within this TRS. It can be seen in the terminal output in Figure 6.6.1 that increasingly large variants are generated by the get variants command for the original term (X ∗ Y).

6.8 Maude: Conclusion and Performance

It has now been proved, through using the Maude tool, that the term rewrite system lacks the finite variant property. Such a constraint shows that variant-based narrowing is not an option to utilize in any unification algorithm for this TRS. Maude's fast performance and ability to seamlessly integrate term rewriting concepts make it a great candidate to use to work with in rewriting systems. However, it's extremely niche nature (the only real documentation on it exists in the form of the manual, as well as an email help list that can be signed up for) makes it hard to easily pick up as there are very few resources on it. Maude also lacks verbose error messages and the syntax in relation to requiring spaced periods after denotes a parser that is rigid and inflexible. Therefore, it is now known that a different approach must be taken to derive a unification algorithm for this TRS which lacks the finite variant property.

7

Deriving a Unification Algorithm Utilizing Prolog: Prolog Basics

The focus of this paper will now be on the Prolog language. After determining variant-based unification would no longer be a viable strategy to perform unification for the quandle TRS, Prolog was determined to be the best language to derive a unification algorithm for the quandle TRS. Prolog is far less of a niche tool than Maude, and thus has far more resources available and an active community of users which make it far more accessible. So what exactly is Prolog? The Prolog language is a logic programming language. It is quite declarative, dynamically typed, and, most importantly, supports built-in syntactic unification. In fact, syntactic unification is one of the defining characteristics of Prolog.

7.1 Facts and Queries

Facts, predicates, and clauses make up the true core of what is considered "pure Prolog". Prolog allows for definitions of "facts", which can be viewed as a pseudo-database system. Facts are predicates which declare some universally quantified (when what the fact is declaring is a variable) truth. Syntactically, facts are an identifier, followed by parentheses which enclose one or more terms. Variables or constants can be passed as the fact's parameters.

Prolog variables begin with an uppercase letter or underscore (variables can be, and often are, singular uppercase letters as well), and Prolog constants begin with a lowercase letter. Facts also must be ended with a period. Consider a simple base of facts that define which division a respective major league baseball team is part of:

```
plays_in(mets,nl_east).
plays_in(phillies,nl_east).
plays_in(braves,nl_east).
plays_in(marlins,nl_east).
plays_in(braves,nl_east).
plays_in(brewers,nl_central).
plays_in(cardinals,nl_central).
plays_in(pirates,nl_central).
plays_in(cubs,nl_central).
plays_in(reds,nl_central).
plays_in(yankees,al_east).
plays_in(rays,al_east).
plays_in(orioles,al_east).
plays_in(red_sox,al_east).
plays_in(blue_jays,al_east).
plays_in(tigers,al_central).
plays_in(twins,al_central).
plays_in(white_sox,al_central).
plays_in(royals,al_central).
plays_in(indians,al_central).
plays_in(al_central,american_league).
plays_in(al_east,american_league).
plays_in(nl_east,national_league).
```

```
plays_in(nl_central,national_league).
plays_in(X,major_league_baseball).
```

Each fact is a different definition of the fact "plays_in", which as a first parameter has the team name, and as the second parameter, the division the team is a part of. There also exist fact definitions for the leagues the divisions are a part of. Finally, the last fact's variable is *universally quantified*. Every team in this fact base, as well as every division, is of course a part of major league baseball.

Therefore since X is a variable, it is declared in this fact every first parameter passed to the plays_in fact is true when the second parameter is major_league_baseball. Now that there exists a fact base, Prolog's querying abilities come into play. First compile the file like so: (Prolog files are denoted by the file extension .pl).

```
consult('file_name.pl').
```

The period is required after every query. Queries allow the user to ask Prolog to verify information about the fact base (or rule, which will be discussed soon). It can be verified that the orioles play in the al_east division by simply entering

```
plays_in(orioles,al_east).
```

```
?- plays_in(orioles,al_east).
true .
```

First, compile the .pl file using consult('file.pl'). This returns true if such a file can successfully be compiled. Prolog also warns about the use of a singleton variable- this is not illegal by any means, but Prolog will throw a new warning every time a new variable is introduced in the program. Prolog is queried whether the fact

```
plays_in(orioles,al_east).
```

is a ground truth. Prolog will consider the facts in the fact base and check if one of the facts matches the query, and if it does, it will return true. But what if it is desired to find

all the teams that play in the al_east division? How could this be done? This is where Prolog's true specialized abilities come into play: it's ability to *backtrack* and *unify*. If there is some fact base, and it is desired to verify multiple facts at once using a singular query, the program can query using a variable.

Variables are existentially quantified, so the presence of a variable in a query signals to Prolog to search for any fact that leads to a ground truth. The user can also signal to continue search the fact base for more facts that lead to a ground truth - known as backtracking. Backtracking erases any unification, so the variable returns to it's non-unified form with every new backtracking solution. The user can signal to Prolog to backtrack to another solution by inputting a semicolon after one solution is found. A semicolon will signal to Prolog to search for another possible solution. The user can continue to press the semicolon key until no more solutions are possible.

```
?- plays_in(Team,al_east).
Team = yankees ;
Team = rays ;
Team = orioles ;
Team = red_sox ;
Team = blue_jays.
```

The fact base is queried with a variable as the first parameter, signaling to Prolog to unify this variable with a constant that lead to ground truth. Then, user input with semicolon signals to Prolog to yet again backtrack and find another possible solution that leads to ground truth- the rest of the teams in the American League East division in the program. If the user had not pressed the semicolon key and simply pressed enter, Prolog would only return one solution, the first fact it encountered that led to a ground truth, which would simply be:

```
Z=yankees
```

7.2 Rules

Prolog has interesting database and querying abilities, but what makes this any more interesting than say, SQL? Well, Prolog also allows for rule definitions. Rules allow for constraints to be the set for the fact base, and to answer questions about it that would be undecideable if only the fact base existed. Rules can be thought of as Prolog's version of functions. Prolog rules can take in some parameters, define some behavior in a body, and then, like all things in Prolog, determine if the body has decided some ground truth.

Rules have a "head", which is identical in structure to the fact definitions, but instead of a period after denoting the end of a definition, have the ":-" symbol. This symbol is considered to represent "logical implication" in Prolog. What this means is that "the head is the logical implication of what was defined in the body". What exists in the body of the rules can be some facts, or some calls to other rules, and these facts or rule invocations imply what was defined in the head. Something that is known to be true, but cannot be considered to be a ground truth when looking at the basic fact base, is that the orioles play in the american_league. They play in the al_east division, and the al_east division is part of the american_league. However, a Prolog query with the structure "plays_in(orioles,american_league)." will return false, because, quite simply, this fact is a *logical implication* of the fact base, but it is not a ground truth. Now it is possible to define a rule that states the logical implication of one team playing in a division that is a part of a broader league would imply that said team also plays in that league. Consider the following definition:

```
part_of_league(X,Y):-
        plays_in(X,Z),
        plays_in(Z,Y).
```

The rule has been defined to take in two parameters that are variables, so Prolog will search it's entire fact base to try and unify the variables with some constants that make

the two facts in the body ground truths. The first argument will be the team, and the second will be the league that the team either does or does not play in. The first thing the rule does is unify the first fact with all ground truths for plays_in(X,Z), with these first ground truths being the facts of a given team playing in a certain division. Z will now have been unified with the division the team X plays in. After this fact there is a comma.

Commas in Prolog represent a logical conjuction, the rule states that for the head to be true, it must be the logical conjuction of the two facts in the body. Next, the body will attempt to unify the plays_in fact with Z as the first argument. Z, from the previous fact, will be a division, and therefore any ground truth for the plays_in fact that has a first argument that is a division will contain a second argument that is a league, either the american or national, since the fact base defines which divisions play in which leagues. The rule then states the following logical implication:

```
A team X plays in league Y iff team X plays in division Z which plays in
↪  league Y
```

Prolog can verify one or more ground truths based on how much backtracking the user defines. This allows for the user to determine all teams that play in a given league like so:

```
?- part_of_league(X,national_league).
X = mets ;
X = phillies ;
X = braves ;
X = marlins ;
X = braves ;
X = brewers ;
X = cardinals ;
X = pirates ;
X = cubs ;
X = reds ;
```

```
false.
```

Prolog also allows for a singular predicate to have multiple clauses. This means it's legal to redefine a rule multiple times, and the program can attempt to see if the logical implication of one clause is true when another turns out to be false. An example of this is if it is desired to know if not only a team is part of a certain league, but if a team plays in major league baseball at all. Of course, this will always be true- this fact was existentially quantified by the fact that all of the teams,divisions, and leagues in the fact base play in major league baseball. The following is what adding a separate clause that declares this to be a logical implication looks like:

```
part_of_league(X,Y):-
        plays_in(X,Z),
        plays_in(Z,Y).

part_of_league(X,Y):-
        plays_in(X,Z),
        plays_in(Z,Y);
        plays_in(X,major_league_baseball).
```

This new definition is identical to the first definition, but includes an extra line. This new clause states that if the second fact cannot be verified to ground truth, for the program to backtrack (this is denoted by the semicolon!) and check if the next fact can be verified to have ground truth. Now querying a rule that previously would be false (like the orioles playing in the national league),would erroneously return true- The program would, after failing the first clause, try the second clause, and see that it can verify the ground truth that the orioles do play in major league baseball.

7.3 Non-Local Control

Non-local control allows for the user to bound Prolog's backtracking abilities and to remove solutions that Prolog would come across by backtracking after finding one original solution. In Prolog, the "!" operator signals that, if the program previously had some rule definition which allows for backtracking, Prolog will not go back and try another solution once it has found one. It's called the cut operator- since it "cuts out" a choice point, and instead makes sure Prolog will only try and satisfy some fact or rule once [16]. An example of this can be seen with a slight re-definition of the previously mentioned plays_in_league clause.

The re-definition of said rule could have been limited to simply have one clause, but for the sake of explaining clause re-definition, it was done in two. Consider what a rule body would look like that says to try and verify the ground truth that a team plays in a league, and if it cannot verify such a fact, to verify that the team plays in major league baseball:

```
part_of_league(X,Y):-
        plays_in(X,Z),
        plays_in(Z,Y);
        plays_in(X,major_league_baseball).
```

Prolog will always backtrack unless it is explicitly told not to- so even if it does verify the plays_in(Z,Y) fact as having ground truth, it will see the semicolon and backtrack to the other choice that stems from the logical conjuction: if it can verify the ground truth the team plays in major league baseball. Seeing this in purely logical terms, what it declared here is the following:

(plays_in(X,Z) $\wedge$ (plays_in(Z,Y) $\vee$ plays_in(X,major_league_baseball)))

However, it seems superflous to have Prolog backtrack to verify that some team plays in major league baseball if it already was able to verify the ground truth that team X plays in some league Y, since it is already known all leagues play in major league baseball. It would seem logical to cut out the backtracking if Prolog was able to verify such a ground

truth. In fact, this can be achieved with the cut operator. Consider the following clause re-definition such that if the program can verify team X plays in league Z, it doesn't have to even verify the truth that team X plays in major league baseball.

```
part_of_league(X,Y):-
        plays_in(X,Z),
        plays_in(Z,Y),!;
        plays_in(X,major_league_baseball).
```

The comma represents logical conjuction, so the ! operator will only ever be reached if the plays_in(Z,Y) fact is verified to have ground truth. Once this is true, the program will encounter the cut (cuts are always evaluated to have ground truth in Prolog) and now it doesn't have to backtrack for any more solutions. If the fact is not verified, the cut will never be hit, but instead hit the semicolon, indicating to the program it should backtrack and try to verify the ground truth that team X plays in major league baseball.

7.4 Lists and Recursion

Lists in Prolog are robust data structures. Prolog lists should not be thought of being similar in any way to arrays or lists in Python, Java, or C. Prolog's lists are heavily dependent on recursion. In fact, Prolog allows only for two legal kind of list initializations. Prolog lists can be initialized to be empty, or allow for head/tail definition [15]. Prolog's lists do not have individual list indices but rather simply a head and a tail. The head of the list could contain just one element, or it could contain numerous elements. It doesn't truly matter: Prolog lists really only have two elements within it, it's head and tail.

The head and tail of a list are defined like so: [Head|Tail]. Therefore to recursively traverse a list, look at the head, and then look at the tail as the new head until finally all that remains is an empty list. Consider what a rule definition that traverses a list and writes to the console each of it's elements would look like:

```
traverse_list([]).
traverse_list([H|T]):-
        writeln(H),
        traverse_list(T).
```

Notice that traverse_list has both a fact and rule definition. The fact definition, which just has an empty list as a parameter, is the "base case". Once the list has been traversed enough times where all that remains is an empty list, the fact is verified as having ground truth, signaling the end of the list. The rule clause shows how lists are inherently recursive data structures. First the head is written out to the console, and then the recursive call is made, this time passing the tail of the list as the parameter. If the list [1,2,3,4] was passed in as an argument, Prolog will see the list in the only structure it knows how to: [Head|Tail], since it's not an empty list. Thus it sees the said list in the structure: [1|2,3,4].

It can be seen then that passing in the tail as the recursive argument, the program will go through a series of calls that looks like this: [1|2,3,4]→ [2|3,4]→ [3|4]→ [4|[ ]]→ [ ]. Ground truth will not occur until the final call, when there is an empty list, indicating the entire list has been traversed.

8

Deriving a Unification Algorithm in Prolog: Developing a Metavariable System

To best be able to implement the rewrite system,it was necessary to develop a metavariable system within the Prolog program. As discussed in chapter 7, Prolog has extremely powerful unification abilities modulo it's variables. Unfortunately, these impressive unification abilities, because of their extremely non-deterministic manner, leads to issues when it comes to being able to match the structure of two terms for narrowing purposes. Being able to utilize metavariables will provide full control over Prolog unification.

8.1 Challenges with Using Prolog Variables to Implement the Rewrite System: Standardizing Apart

To implement the rewrite system in Prolog, a prefix representation of the binary $*$ and / operators were utilized. Using Prolog variables, there also was the infix "rewriting" operator, which would separate the redex and reduct of a rewrite rule. This operator has the form of "==>". The rewrite rules will have the following structure when attempting to use Prolog's built in variables:

```
star(X,X) ==> X.
divi(X,X) ==>  X.
```

```
divi(star(X,Y),Y) ==>  X.

star(divi(X,Y),Y) ==>  X.

star(X, star(Y,Z)) ==> star(star(divi(X,Z),Y),Z).

divi(X, star(Y,Z)) ==> star(divi(divi(X,Z),Y),Z).

star(X, divi(Y,Z)) ==> divi(star(star(X,Z),Y),Z).

divi(X, divi(Y,Z)) ==> divi(divi(star(X,Z),Y),Z).
```

Consider some predicate, *unify*, that would search for some unifier and rewrite sequence to unify two terms. Such a predicate also would have to ensure that the two terms that will be unified are not *standardized apart*. What this means is that Prolog's variables are truly logical variables, and therefore are seen merely as placeholders for some arbitrary value. Standardizing Prolog's variables apart means declaring any two variables not in the same term signals to Prolog that the language can instantiate them to any such placeholder value. To Prolog, all placeholder values are considered to be *variants* of each other, meaning they have equal term structure. Variant equality in Prolog is encoded by the =@= operator. Since all Prolog variables are logical variables, they are seen as having variant equality modulo their roles as placeholders [16]. Therefore, the query:

$$\text{A=@=B}$$

is seen as having ground truth, as A and B are both variables, and therefore structurally equivalent placeholders regardless of their differing identifiers.

Returning to the discussion of standardizing apart in a hypothetical "unify" predicate, if there was such a predicate that took in two arguments, the first being the first term, and the second term that would be the target to unify with, the variables would be seen as being standardized apart by Prolog. Therefore, passing in two arguments to a unification predicate will be seen by Prolog merely as placeholders, while the information that must be known is the structure of the terms. Being able to successfully identify term structure is essential to the rewriting system because of the need to successfully perform narrowing.

As discussed in previous chapters, narrowing is the ability to make a term a structure's equivalent to a redex of the a rewrite system. Therefore to successfully narrow, it would be necessary to check variant equality in Prolog. Consider a term, X∗Y. Such a term could be narrowed to either an idempotency redex, or a delta rule redex. This new redex can then be rewritten to it's respective reduct, known as a rewriting step. All narrowing steps will have a corresponding rewriting step, as the goal of narrowing is to transform a term into a redex, and all redexes have a corresponding reduct that they are rewritten to. The following shows the process of narrowing and then rewriting to a new term:

```
Narrowing to Idempotence Rule:
Narrowing Step: Substitute Y for X
New Term: X * X
Rewrite to Corresponding Idempotence Reduct:
X * X ==> X
New Term After Narrowing and Rewriting Steps: X

Narrowing to Delta Rule Redex:
Narrowing Step: Substitute Y for A * B
New Term: X * (A * B)
Rewrite to Corresponding Reduct:
(((X / B) * A) * B)
```

However, as mentioned previously, Prolog sees all of it's variables as having equal value modulo variant equality. If two terms were to have the same structure yet be composed of different variables, Prolog would still see them as having structural equality. Consider the hypothetical unify predicate then does not take in two standardize apart terms, but simply one standardized together term like so:

unify(TERM1==>TERM2)

which indicates the predicate will attempt to unify TERM1 with TERM2. However, both of the terms are considered to be part of a singular term (connected by the ⟩ symbol) and therefore are standardized together. While previously TERM1 and TERM2 were standardized apart like so:

unify(TERM1,TERM2)

the variables within TERM1 and TERM2 would have no relation to each other. Even if two terms were to be passed in like so:

unify(star(X,X),X)

the idempotent rule should be able to unify immediately the two terms immediately. Prolog, however, would not see the X in argument two as being the same value as the two X's in argument 1, indicating once again Prolog's interest in representing variables merely as placeholders.

8.2 Challenges with Using Prolog Variables to Implement the Rewrite System: Context-Based Narrowing

For any successful unification algorithm, the program must be able to successfully narrow some term or subterm. Narrowing, as mentioned in previous chapters, is the process of attempting to match some term to a redex of a rewrite rule so said term can be transformed to it's respective reduct. *Context-based* narrowing is a narrowing strategy different from the variant narrowing strategy mentioned in the Maude chapter. Context-based narrowing sees each individual subterm of a term as a "context" which can be transformed to some other term through some narrowing and rewriting step. Contexts are instances of the original term in which a subterm has been substituted out for a placeholder value that will be filled by some rewritten version of the subterm. Contexts are viewed as pairs, in which the two elements of the pair are the context, and the subterm that was substituted out to form said context. Consider the following two terms:

star(star(X,Y),star(A,B))

The subterm for which a context will be formed is star(X,Y). The placeholder value for all contexts will simply be referred to as "hole". Therefore, the context pair for the star(X,Y) subterm is:

((star(hole,star(A,B)),star(X,Y)))

Furthermore, as mentioned in the previous section, Prolog's ability to check variant equality would be helpful to form contexts from specific subterms. However, the fact that Prolog sees all of it's respective variables, standardized together or not, as having equal variant equality, throws a wrench in the whole process. Using Prolog variables in combination with the variant equality operator leads to Prolog seeing the two subterms star(A,B), and star(X,Y) as still being variants of each other. This is because they both share the same operator, and have equivalent arity, and it's arguments do not share the same identifier. Prolog's =@= operator may see the following as false:

star(X,Y) =@= star(X,X)

because the second term's structure indicates that both arguments to the operator are equivalent, but it will see the following as true:

star(X,Y) =@= star(A,B)

Therefore, it would be impossible to form the "holes" needed for context-based narrowing as Prolog's variant equality method would not be able to differentiate a hole in the star(X,Y) subterm and a hole in the star(A,B) subterm.

Prolog's built-in unification abilities here would unify X with A and Y with B, thus unifying two terms in a situation where unification is not desired. The issue lies in Prolog's usage of logical variables, in which unification is always possible between two terms that have an equivalent structure, even if the variables themselves that make up the term are syntactically not equivalent. This issue led to the creation of a metavariable system that is able to bypass this issue.

8.3 Building the Metavariable System and Re-Defining the Rewrite Rules

As discussed in the previous section, Prolog's built-in variables could not be used to build the rewrite system. Instead of these built-in variables, a system of *metavariables*, or user-defined variables that would exist within the Prolog environment, were defined. The metavariables are defined by the identifier v, followed by a numerical value. The numerical value that follow the "v" prefix are what gives each metavariable a unique value. Thus, metavariables are defined like so in the program:

```
Valid Metavariables: v(0),v(1),v(2),v(3)..
```

Previously, rewrite rules were denoted by the $==\rangle$ rewriting operator. These rules were defined one time each. However, for the purposes of making narrowing a real possibility within the program, the rewrite rules would now be defined in terms of *narrowing relations*. A narrowing relation is a possible way a term structure of redex and reduct could relate to one of the rewrite rules through some narrowing step. Consider a simple example of this. Firstly, all rewrite rules are now defined through two narrowing relations: Either through some narrowing step which requires a substitution, or if the redex and reduct already exactly matches the structure of a rewrite rule, meaning no substitution is needed to invoke the rewrite rule. Thus, each of these rules would be defined by the following structure, either as in fact or predicate form:

rewrite_rule(REDEX,REDUCT,SUBSTITUTION)

The first two arguments are rather self explanatory in nature, and the third argument represents a substitution that was made to get the original term to be rewritten to the redex. In other words, it is the narrowing step required. Therefore, in the fully narrowed rule or predicate definitions for a rewrite rule, the third argument is simply an empty list. For rule definitions where a substitution is required, and list is not null, the substitution list exists in the form:

[v(X)=VALUE]

Where v(X) can represent any metavariable, and VALUE can be either another metavariable, a constant, or an entire term. Now that the new structure of rewrite rule definitions have been explained, consider what this would look like for the idempotence rules, for which there exist 6 facts and 4 predicates which utilize 4 of the facts defined:

```
idempotent_narrow_1(star(T1,T2),T1,[T2=T1]).
idempotent_narrow_2(star(T1,T2),T2,[T1=T2]).
idempotent_narrowed(star(T1,T1),T1,[]).
idempotent_narrow_3(divi(T1,T2),T2,[T1=T2]).
idempotent_narrow_4(divi(T1,T2),T1,[T2=T1]).
idempotent_narrowed_2(divi(T2,T2),T2,[]).

//Caller Predicates Which Enforce Not Substituting Out Non-Variable
↪  Subterm:

idem_narrow_1_rule(REDEX,REDUCT,SUB):-
        idempotent_narrow_1(REDEX,REDUCT,SUB),
        get_lpos_term(SUB,X),
        metavar(X).

idem_narrow_2_rule(REDEX,REDUCT,SUB):-
        idempotent_narrow_2(REDEX,REDUCT,SUB),
        get_lpos_term(SUB,X),
        metavar(X).

idem_narrow_3_rule(REDEX,REDUCT,SUB):-
        idempotent_narrow_3(REDEX,REDUCT,SUB),
```

```
        get_lpos_term(SUB,X),
        metavar(X).


idem_narrow_4_rule(REDEX,REDUCT,SUB):-
        idempotent_narrow_4(REDEX,REDUCT,SUB),
        get_lpos_term(SUB,X),
        metavar(X).
```

The lpos/2 predicate will unify the second argument with the left hand side of the substitution that exists in coordinate one. The metavar/1 predicate checks if the argument passed to it is in fact a user-defined metavariable. An example of the usage of the predicates it thus:

```
?- get_lpos_term([v(0)=star(v(1),v(2))],Term_To_Replace).
   Term_To_Replace = v(0).


?- metavar(v(1)).
   true.
```

The facts ending -ed, in this case, idempotent_narrowed and idempotent_narrowed_2 indicate when the redex and reduct of a term match exactly to the structure of an idempotence rule: the two arguments passed to the operator of the redex are exactly the same as the reduct. The final argument is an empty list, indicating that no narrowing step is needed to use the respective rewrite rule, hence the "fully_narrowed" identifier. This fact only succeeds when passed a redex and reduct that match exactly to a the idempotent rewrite rules. The third argument passed in to such a fact, if it were to be queried, would simply unify with an empty list.

```
?- idempotent_narrowed(star(v(0),v(0)),v(0),Unifier).
   Unifier = [].

?- idempotent_narrowed(star(v(0),v(1)),v(0),Unifier).
   false.
```

The rest of the fact definitions represent an invocation of the rewrite rule that requires a narrowing step. For a narrowing step to an idempotence rule to exist, there must be two arguments to the operator in the redex, and the existence of one of those arguments as the reduct. Therefore, the substitution required to match some term to the idempotent rewrite rule is a substitution which make both arguments of the redex equivalent to the reduct, thus matching the original term to the exact structure of the rewrite rule. To make this possible, four rule definitions were needed to account for all of the cases in which the reduct is equivalent to either the first or second argument of the redex. The four rule definitions consist of two subcases each nested within the two cases of the term either being rooted at the star or divi operator. The four cases are then:

```
Case 1: The term is rooted at the star operator, and the term is narrowed
 ↪  to the idempotent redex by substituting the original left subterm for
 ↪  the right subterm so both subterms are equivalent
Case 2: The term is rooted at the star operator, and the term is narrowed
 ↪  to the idempotent redex by substituting the original right subterm
 ↪  for the left so both subterms are equivalent
Case 3: The term is rooted at the div operator, and the term is narrowed
 ↪  to the idempotent redex by substituting the original left subterm for
 ↪  the right subterm so both subterms are equivalent
Case 4: The term is rooted at the div operator, and the term is narrowed
 ↪  to the idempotent redex by substituting the original right subterm
 ↪  for the left subterm so both subterms are equivalent
```

The four caller predicates ensure that the substitution being made to narrow is not one that substitutes out an entire term for a metavariable as substitutions can only be made at variable positions. The get_lpos_term predicate simply pulls out the left hand side of a substitution. The basic structure of a substitution is one of

[Variable to Sub Out=Replacement]

Therefore, the only thing that should exist as the left hand side of an argument is a metavariable- which is what the next predicate "metavar" checks.

The following example shows through querying why the 4 predicates are needed, and their usage:

```
?- idem_narrow_3_rule(star(v(0),v(1)),v(1),SUB).
SUB = [v(0)=v(1)].


?- idem_narrow_2(star(star(v(0),v(1)),v(2)) v(2) SUB).
SUB = [star(v(0),v(1))=v(2)].


?- idem_narrow_2_rule(star(star(v(0),v(1)),v(2)) v(2) SUB).
false.
```

The first query here uses the idempotent narrow 3 fact, whose structure states the the first two arguments can be narrowed to an idempotence rule unifying the first argument of the redex with the second since the reduct is equivalent to the second argument of the redex. However, the constraint must also hold that substitution can only occur at variable positions. This rule, without the constraints of it's respective predicate, allows for a narrowing step which substitutes out a non-variable position for a variable. The next query shows such a call to the caller predicate for the third idempotent narrow fact fails, as the left hand side of the substitution is not a metavariable. Finally, the third query simply shows what a successful call to one of the caller predicate looks like. There is a

redex, (v(0)/v(1)), and a reduct, v(1). The left hand side of the term can be unified with the right hand side by invoking the idempotence rule by substituting v(0) for v(1).The right-cancellation rules are built in the same way as the idempotence rules.

Narrowing to a right cancellation rule differs from narrowing to a idempotence rule as now the redex structure that will be matched is:

(A*B)/B, or (A/B)*B, where both reducts are A

Therefore, to narrow to an invocation of these rewrite rules, the redex must be rooted at the opposite operator of it's left subterm, and it's reduct must be equivalent to the left hand side of it's left subterm. If these conditions are met, then such a substitution could be made which unifies the right hand side of the left subterm and the argument that existed as the right subterm of the redex. The respective predicates exist for each of the rule definitions that require a non-empty substitution to ensure the same constraint that exists within the idempotence predicates: to avoid substituting out a non-variable:

```
right_cancel_narrow_1(star(divi(T1,T2),T3),T1,[T2=T3]).
right_cancel_narrow_2(star(divi(T1,T2),T3),T1,[T3=T2]).
right_cancel_narrowed_1(star(divi(T1,T2),T2),T1,[]).
right_cancel_narrow_3(divi(star(T1,T2),T3),T1,[T2=T3]).
right_cancel_narrow_4(divi(star(T1,T2),T3),T1,[T3=T2]).
right_cancel_narrowed_2(divi(star(T1,T2),T2),T1,[]).

//Caller Predicates Which Enforce Constraint of Not Substituing Out a
↪  Non-Variable Subterm:

right_cancel_narrow_rule_1(REDEX,REDUCT,SUB):-
        right_cancel_narrow_1(REDEX,REDUCT,SUB),
        get_lpos_term(SUB,X),
```

```
        metavar(X).


right_cancel_narrow_rule_2(REDEX,REDUCT,SUB):-
        right_cancel_narrow_2(REDEX,REDUCT,SUB),
        get_lpos_term(SUB,X),
        metavar(X).


right_cancel_narrow_rule_3(REDEX,REDUCT,SUB):-
        right_cancel_narrow_3(REDEX,REDUCT,SUB),
        get_lpos_term(SUB,X),
        metavar(X).


right_cancel_narrow_rule_4(REDEX,REDUCT,SUB):-
        right_cancel_narrow_4(REDEX,REDUCT,SUB),
        get_lpos_term(SUB,X),
        metavar(X).
```

The following example shows some narrowing modulo these rules:

```
?- right_cancel_narrow_rule_1(star(divi(v(0),v(2)),v(3)) v(0) SUB).
SUB = [v(2)=v(3)].
?- right_cancel_narrow_rule_1(divi(star(v(0),v(1)),v(2)) v(0) SUB).
SUB = [v(1)=v(2)].
```

8.4 Introducing New Metavariables: Defining the Delta Rules

It should once again be noted what the delta rules' structures are, using the metavariables:

```
v(0) / ( v(1) / v(2) ) = ( ( v(0) * v(2) ) / v(1) ) / v(2)
v(0) * ( v(1) / v(2) ) = ( ( v(0) *  v(2) ) * v(1) ) / v(2)
```

```
v(0) * ( v(1) * v(2) ) = ( ( v(0) / v(2) ) * v(1) ) * v(2)
v(0) / ( v(1) * v(2) ) = ( ( v(0) / v(2) ) / v(1) ) * v(2)
```

Furthermore, as denoted in chapter 6, such a narrowing relation exists in this TRS wherein every term of the structure:

```
v(0) BINARY_OPERATOR v(1)
```

can be narrowed to any given delta rule redex through the narrowing step:

```
[v(1)-> (v(2) BINARY_OPERATOR v(3))]
```

It's important to note that this substitution passes the occurs check. The v(1) metavariable cannot exist in the right hand side of the substitution, as this leads to an endless substitution cycle that fails the occurs check. Thus, it is necessary to introduce two entirely new metavariables in the right hand position of a substitution in a narrowing relation for a delta rule. Therefore, to define the delta rule narrowing relation within the context of the metavariable, it is required to introduce two new variables. This can be achieved through Prolog's built-in assert and retract predicates.

8.5 Assert and Retract

The assert and retract predicates in Prolog are used to load and unload facts to a given program's dynamic database. Assertion takes in as a parameter a fact definition. Given this fact definition defines a ground truth, any query to this fact will return the ground truth. It is also important to note any fact defined in the original database of a program is not used in a call to assert. Consider the following example of the assertion of some fact, "fun", with the ground truth "sproj". Consider the following example, performed in the SWI-Prolog Terminal:

```
?- assert(fun(sproj)).
true.
```

```
?- fun(X).
X = sproj.

?- retract(fun(X)).
X = sproj.

?- fun(X).
false.
```

It can be seen here that once fun() is asserted to have the ground truth "sproj", a query to fun() unifies the variable X with the constant "sproj". Furthermore, once retract is called, fun no longer has the ground truth value of "sproj". A further query simply returns as false, since this query was not defined elsewhere and it's ground truth no longer is defined.

The bindings defined through assert and retract exist in the "dynamic database" of a program, while those bindings defined by the user in a program exist in the "static database" [15]. It is also important to note assert and retract should NEVER be accessing facts that were defined in the original static database of the program. There also exists another predicate, retractall, which simply does the same job as retract, but retracts all possible ground truths (as it is possible to assert multiple ground truths for a singular fact).

8.6 Using Assert and Retract To Build the Delta Rule Narrowing Relation

Defining the delta rule narrowing relation requires the introduction of two new variables. In the Prolog unification program, a call to the delta rule narrowing predicate is able to achieve this through the following methods:

```
Method to Introduce Two New Metavariables For a Delta Rule Reduct:
        1. Take in term that will be narrowed
        2. Get largest metavariable index of said term
        3. Save and assert next largest metavariable index
        4. Retract current metavariable ground truth (the next largest
        ↪  index in relation to the metavariable indices in the term)
        5. Assert and save binding to next largest integer value (largest
        ↪  metavariable index of term + 2)
        6. Build reduct of delta rule with two metavariables that have
        ↪  indices equivalent to the saved value from the assertions
```

As an example,consider a narrowing relation from the term star(v(0),v(1)) to a delta rule redex star(v(0),star(v(1),v(2))) and it's consequential rewriting:

```
        1.star(v(0),v(1))
        2. Largest metavariable index: 1
        3. Assert v(2), and save what is now the ground truth of v, which
        ↪  is 2.
        4. Retract v(2)
        5. Assert v(3), and save a binding to what is now the ground
        ↪  truth of v, which is 3
        6. Construct the new term:
        v(0) * ( v(1) * v(2) ) = ( ( v(0) / v(2) ) * v(1) ) * v(2)
```

The narrowing relations for delta rules that require some introduction of new metavariables perform all of these steps. Here is the predicate for delta rule where the topmost operator is a star and the right subterm is also rooted at a star. It involves a fact definition, and a rule definition which will utilize said fact:

```
delta_rule_narrowed_star_star_new_vars(star(T,v(N)),
star(star(divi(T,v(J)),v(M)),v(J)),[v(N)=star(v(M),v(J))]).


delta_rule_1(star(T,v(N)),RDCT,SUB):-
        RDCT \=@= T,
        RDCT \=@= v(N),
        get_largest_variable_index(star(T,v(N)),B),
        next_variable_index(D),
        NEXT_INDEX is D+1,
        delta_rule_narrowed_star_star_new_vars(star(T,v(N)),
        star(star(divi(T,v(NEXT_INDEX)),v(D)),v(NEXT_INDEX)),
        SUB),
        get_lpos_term(SUB,X),
        metavar(X),
        retractall(v(_)),
        assert(v(NEXT_INDEX)),
        copy_term(star(star(divi(T,v(NEXT_INDEX)),v(D)),v(NEXT_INDEX)),RDCT).
```

The first fact, ending with the suffix "new vars", just defines the narrowing relation for transforming a term of the structure star(T,v(N)) to a delta rule reduct, along with the substitution required to make this rewriting possible. The first two facts within the rule ensure that the delta rule cannot clash with a redex that matches to an idempotence rule. The get_largest_variable_index predicate asserts the largest variable index of the given redex in the dynamic database, and the next_variable_index predicate asserts the next largest integer value in the v() metavariable fact, as well as returning that variable index to the user in variable D. The NEXT_INDEX value is the current largest metvariable index plus one. The line after constructs the new reduct of the delta rule with the new metavariable indices in both the reduct and the substitution. The following two lines

ensure what was substituted out was indeed a metavariable, and not a compound term or constant. Finally, the new metavariable index, the original largest index + 2, is asserted in the dynamic database, and the reduct of the "new vars" fact is copied to the RDCT argument of the predicate. Finally, the following output is what occurs after querying said rule definition:

```
?- delta_rule_1(star(v(0),v(1)),RDCT,SUB).
RDCT = star(star(divi(v(0), v(3)), v(2)), v(3)),
SUB = [v(1)=star(v(2), v(3))].
```

Similar rule/fact pairs are defined for all possible narrowing relations of the delta rule that require the introduction of new metavariables. Of course, there also exist the cases in which no substitution is required to narrow a term to a delta rule redex. These require no new metavariable introductions, and are the following fact base:

```
delta_rule_narrowed_1(star(T1,star(T2,T3)),star(star(divi(T1,T3),T2),T3),[]).
delta_rule_narrowed_2(star(T1,divi(T2,T3)),divi(star(star(T1,T3),T2),T3),[]).
delta_rule_narrowed_3(divi(T1,star(T2,T3)),star(divi(divi(T1,T3),T2),T3),[]).
delta_rule_narrowed_4(divi(T1,divi(T2,T3)),divi(divi(star(T1,T3),T2),T3),[]).
```

A query to the fact ending with suffix 2, the delta rule rooted at the star operator, and whose right subterm is rooted at the divi operator, behaves like so:

```
?- delta_rule_narrowed_2(star(v(0),divi(v(1),v(2))),RDCT,SUB).
RDCT = divi(star(star(v(0), v(2)), v(1)), v(2)),
SUB = [].
```

SUB here is an empty list since no narrowing is required to match this redex to the delta rule redex- their structures are already identical.

9

The Algorithm for Quandle Unification

Using the metavariable system as discussed in the previous chapter, it became possible to formulate a terminating unification algorithm. The algorithm was able to successfully generate the most general unifiers for a certain number of examples, as well as determine when the examples had no existing unifier. A most general unifier is a unifier such that it is possible to construct every other unifier for the two expressions from this original unifier by composing it with some other substitution. In other words, for any unifier ϕ for two terms, there exists a unifier sigma such that $\sigma\epsilon{=}\phi$ [14]. The epsilon in this equation can be some other substitution that can be composed with the most general unifier, σ, to create any other unifier in the set of possible unifiers, ϕ.

9.1 Unification: Simple Cases

The simple cases for which to handle unification were those in which the two terms were either one narrowing step away or the two terms already exactly matched the structure of the redex and reduct and a rewrite rule. In this case, their unifier is simply an empty list, as no narrowing is required. In this section, the symbol "=?" will denote a question of equational unification. An example of the case of no unifier being required is such:

```
(v(0) * v(0)) =? v(0)
```

In this case, the left term and right term match exactly to the structure of the redex and reduct of the idempotency rule for the star operator. There is also the case of a term only being one narrowing step away at it's topmost operator from unifying with the other term. An example of this case would be:

```
(v(0) * v(1))) =? v(0)
```

In this case, substituting v(1) for v(0) leads to the left side being equivalent to the redex of the idempotency rule. Note also that the substitution for the left hand side occurs at the topmost operator, the operator where the overall left and right hand subterms are rooted. Thus, the simple cases exist in two domains:

```
1. No unifier is needed: The terms that will be unified are already
↪  identical to the redex and reduct of a rewrite rule
2: A unifier exists such that:
        A. The substitution is made at the topmost level
        B. Only one narrowing and rewriting sequence is needed to
        ↪  transform one of the terms to the other
```

As noted in the previous chapter, all of the rewrite rules have been transformed to fit the structure of a narrowing relation. Thanks to Prolog's backtracking abilities, the program can attempt all possible narrowing relations on the terms. As mentioned in the last chapter as well, there are two distinct domains of narrowing relations for the rewrite rules in the program:

```
1. Some narrowing step is required: A substitution is needed to transform
↪  the term in coordinate 1 into the redex of the given rewrite rule to
↪  match the reduct  that exists in coordinate 2. The substitution to
↪  make this possible exists in coordinate 3.
2. No substitution/narrowing is required as the term in coordinate 1
↪  already matches the structure of the redex of the reduct in
↪  coordinate 2. The substitution in coordinate 3 is simply an empty
↪  list.
```

Therefore, all narrowing relations that require some substitution are defined in the predicate narrow_step_all. This predicate attempts to match the term in coordinate 1 to some redex of one of the rewrite rules and then unify the argument of coordinate 3 with the substitution required to make this possible. It is constructed like so:

```
narrow_step_all(REDEX,REDUCT,SUBSTITUTION):-
        idem_narrow_1_rule(REDEX,REDUCT,SUBSTITUTION);
        idem_narrow_2_rule(REDEX,REDUCT,SUBSTITUTION);
        idem_narrow_3_rule(REDEX,REDUCT,SUBSTITUTION);
        idem_narrow_4_rule(REDEX,REDUCT,SUBSTITUTION);
        right_cancel_narrow_rule_1(REDEX,REDUCT,SUBSTITUTION);
        right_cancel_narrow_rule_2(REDEX,REDUCT,SUBSTITUTION);
        delta_rule_1(REDEX,REDUCT,SUBSTITUTION);
        delta_rule_2(REDEX,REDUCT,SUBSTITUTION);
        delta_rule_3(REDEX,REDUCT,SUBSTITUTION);
        delta_rule_4(REDEX,REDUCT,SUBSTITUTION).
```

The predicate will backtrack through all possible narrowing relations. Of course, there also is the case of two terms that already exactly match the structure of the redex and reduct of a rewrite rule. The predicate to check for this case is nearly identical to the previous predicate, except all of the rules in this predicate have an empty substitution and the

redex and reduct are already identical to the redex and reduct of their respective rewrite rule.

```
narrow_step_fully_rewritten(REDEX,REDUCT,SUBSTITUTION):-
        idempotent_narrowed(REDEX,REDUCT,SUBSTITUTION);
        idempotent_narrowed_2(REDEX,REDUCT,SUBSTITUTION);
        right_cancel_narrowed_1(REDEX,REDUCT,SUBSTITUTION);
        right_cancel_narrowed_2(REDEX,REDUCT,SUBSTITUTION);
        delta_rule_constants_1(REDEX,REDUCT,SUBSTITUTION);
        delta_rule_constants_2(REDEX,REDUCT,SUBSTITUTION);
        delta_rule_constants_3(REDEX,REDUCT,SUBSTITUTION);
        delta_rule_constants_4(REDEX,REDUCT,SUBSTITUTION).
```

Finally, it is possible that either one of these simple cases could be correct for any two given terms. Therefore, the formal predicate for unification should exist like so:

```
unify(Term1, Term2, Unifier):-
narrow_step_all(Term1, Term2, Unifier);
narrow_step_fully_rewritten(Term1,Term2,Unifier).
```

Now using this predicate is is possible to unify both of the aforementioned simple cases in this chapter.

```
?- unify(star(v(0),v(1)),v(1),Unifier).
Unifier = [v(0)=v(1)].


?- unify(star(v(0),v(0)),v(0),Unifier).
Unifier = [].
```

The unifier in the second case is of course empty as the two terms are already identical to the redex and reduct of the idempotence rule. Therefore, this approach of considering only a singular narrowing step at the topmost operator works for works for the simple cases.

However, this approach is not nearly sufficient: There are two critical aspects of it that are missing, being that it cannot narrow at subterms, and it doesn't take into account any narrowing sequence that takes more than one narrowing step. Furthermore, this poses the further question of how to avoid the endless narrowing sequences caused by the lack of the finite variant property. In the rest of the chapter, a more complete approach to these more complex issues will be discussed.

9.2 Reducing the Unification Problem

Any two unifiable terms are still unifiable in the form

```
a * Term1 = a * Term2
```

where a is any constant that does not appear in either Term1 or Term2. The structure of the right cancellation rules in the rewrite system makes this possible. Consider the simple case of unifying two different variables, v(0) and v(1). Intuitively, there exist two unifiers: []v(0) = v(1)] or []v(1) = v(0)]). Consider the "transformed" problem, which is now

```
a * v(0) =? a * v(1)
```

Much like other algebraic problems that use variables, the problem can be altered such that all variables are moved to one side of the equation, leaving only a singular constant on the other. This is a *matching problem*. Unification is the problem of solving equations of terms by finding substitutions that make the terms equal. Matching is the problem of solving equations where only one of the terms is altered by the substitution. [17]. Therefore, what has just occurred was a reduction from an equational unification problem (E-Unification) to the ostensibly easier matching problem (E-matching).

In this case, all of the variables can be moved to one side by div'ing both sides by v(1). This leaves the structure:

```
(a * v(0))/v(1) =? (a * v(1)) / v(1) =>
(a * v(0))/v(1) =? a
```

The v(1) on the right side was eliminated by the fact the term became equivalent in structure to a right cancellation rule, leaving only a. The left hand side term can also be transformed now to a by an invocation of the right cancellation rule as well, by setting either variable equal to each other:

```
Unifier: [v(1)=v(0)]
(a * v(0))/v(1) =? a ->
(a * v(0))/v(0) =? a ->
a =? a
```

```
Unifier: [v(0)=v(1)]
(a * v(0))/v(1) =? a ->
(a * v(1))/v(1) =? a ->
a = ? a
```

Therefore, any reduced unification problem now boils down to checking if Term1 can be made equivalent to Term2 such that a right cancellation rule can be invoked making a = a. It should be noted that if the unification problem presented is already in the form of a matching problem, no further modification of the terms are required.

9.3 Unification Algorithm: Topmost Narrowing

Going step by step then, the two first steps of the unification algorithm have already been covered in this chapter. These steps are to check if the unification problem exists in the domain of the "simple" cases denoted in section 1, and if no unifier is found through this method, to proceed to "transform" the terms as explained in section two. After this step, one side of the unification problem will include all of the variables/constants, and the other will simply be "a" (or some other constant, depending if a appears in any of the original terms) Once these preliminary steps have been executed, the next step is to take a

narrowing step at the topmost level. It should once again be noted that "topmost" means a narrowing step that attempts to narrow the entire term to a redex. To better illustrate the progression of this algorithm, consider the following unification problem:

```
(((a/c) * b) * c)/v(0) =? a
```

In this case, the first two steps will be skipped, as both sides, or terms that are going to be unified, cannot be unified in a single narrowing step, and are already in the "transformed" structure of step 2. The next step is to narrow the overall term, and it is possible to narrow the overall term that is non-normalized to a delta rule redex like so, leading to the following narrowing and rewriting sequence:

```
Substitution: [v(0)=v(1) * v(2)]
Narrowed Term: (((a/c) * b) * c)/(v(1) * v(2))
Rewritten Term:
(((((a/c) * b) * c)/v(2))/v(1))*v(2)
```

The overall term was narrowed to match the structure of the delta rule whose redex is (X / (Y * Z)). As mentioned in chapter 6, it is always possible to take a narrowing step to a delta rule redex by making a substitution at the right subterm.

Finally, the term is then transformed to the corresponding reduct of the redex it was narrowed to. It should be noted Prolog's backtracking abilities will also allow it to narrow to an invocation of the idempotence rule, but as the algorithm progresses, it will become clear this is an incorrect step that Prolog will eventually discard as a viable narrowing step.

9.4 Unification Algorithm: Delta Rule Bounding and Subterm Narrowing

The next stage of the unification algorithm covers the narrowing and rewriting that will occur at the subterms of the overall term. Before attacking this stage of the algorithm, there is an ancillary step. This is to bound the usage of the delta rules at the subterm

narrowing stage. As discussed in chapter 6, the quandle TRS lacks the finite variant property due to the presence of the delta rules, and therefore to avoid the possibility of endless rewriting sequences, normalization must be induced by bounding the usage of these rules. The bound that was chosen for the maximum number of delta rule invocations for was truncate($\log_2 X$), where X was the number of operators that existed in the term after the topmost narrowing step occurred.

9.5 Log Base 2 Bounding: Explanation

Consider a unification problem of the structure:

$$(a * (b * (c * d))) \,/\, v(0) =? \, a$$

It is possible to unify the two terms by narrowing to a right cancellation redex, leading to a most general unifier of [v(0) = (b ∗ (c ∗ d))]. However, if this problem were to be restated such that the left hand subterm, (a ∗ (b ∗ (c ∗ d))), was normalized by narrowing to a delta rule redex, the new unification problem would exist in the form:

$$(((((((a \,/\, d) \,/\, c) * d) * b) \,/\, a) * b) * d) \,/\, v(0)$$

Owing to the fact convergence was proved for the quandle TRS in chapter 5, the same most general unifier is found, since the order of the rewrite steps taken is irrelevant and the same normal form will still be reached. However, it would now take two delta-rule narrowing steps to eventually obtain it. It has been seen that the usage of the delta rule narrowing steps comes into play heavily within unification problems where there is a left hand subterm that exists in the form:

$$t_n = \begin{cases} a_0, & n = 0 \\ a_n * t_{n-1}, & n > 0. \end{cases} \tag{9.5.1}$$

For a unification problem in the form $t_{n+1}/v(0) = a_{n+1}$, the obvious most general unifier is [v(0)=t_n], by narrowing to a right cancellation redex. However, if the same problem is considered except that t_{n+1} has been normalized by the usage of a delta rule narrowing step, it will now take n delta rule narrowing steps to find the most general unifier [v(0) = t_n].

It was stated in the McGrail paper [4] that the run time of any algorithm to find the normal form for a quandle term must be $\Omega(2^n)$. The justification for this is that for any term t with k operators, the normal form (nf) of t will have $\Omega(2^k)$ operators. For n>0, the normal form of the term t_n/v(0) will have about 2^n operators. Consider the original example provided in this section, the term:

$$(a * (b * (c * d)))/v(0)$$

When a topmost narrowing step is taken by narrowing to a delta rule, and then the left hand subterm is narrowed to a delta rule as well, the new term is:

$$((((((a / (c * d)) * b) * (c * d)) / v(2)) / v(1)) * v(2))$$

This term required two delta rule steps to expand out into this form, and the most general unifier can be found now by continuous applications of the right cancellation rule. It will take about $log_2\ k$ delta rule narrowing steps, where k is the number of operators of the normal form, to find the most general unifier [v(0) = t_n]. This bound will be treated as the universal upper bound on the number of delta rule narrowing steps that can be taken in any given rewrite sequence within any rewriting sequence during the algorithm.

9.6 Algorithm Execution

The bound on delta rule invocations for the term:

```
(((((a/c) * b) * c)/v(2))/v(1))*v(2)
```

will be truncate($\log_2 6$), which equals 2. It should be noted that there already has been one usage of the delta rules: at the topmost step, so therefore there remains only one legal usage of the delta rule that could still be utilized in this rewriting sequence at any point. The TRS without the delta rules becomes an idempotent right quasigroup (IRQ) than a quandle, and thus omitting the delta rules for rewriting after a certain point re-introduces the finite variant property [1].

The "transformation" step introduces a constant as the prefix of the original term. Thus, if the two original terms indeed were able to be cancelled out by the right cancellation rule, it would be proved they could be made equivalent to each other. This would leave only the constant at the prefix, and all singular constants are of course normalized since no further narrowing or rewriting can occur at a singular constant. Now that a delta rule bound has been established, the subterm narrowing can commence. Subterm narrowing works in an "inner-to-outer" nature. Consider the term that is being unified to "a", and it's "inner-to-outer" subterms:

```
Term: (((((a/c) * b) * c)/v(2))/v(1))*v(2)
Inner To Outer Subterms: (a/c), ((a/c) * b), (((a/c) * b) * c), (((a/c) *
↪  b) * c)/v(2), ((((a/c) * b) * c)/v(2))/v(1), (((((a/c) * b) *
↪  c)/v(2))/v(1))*v(2)
```

The ordering of inner to outermost subterms proceeds from the most nested subterm of the left subterm outwards until the overall, "topmost" term is reached. It can be seen that recursively taking the left subterm of the left subterm eventually reaches a point where no more non-variable subterms can be found. When this base case has been hit, the "innermost" subterm has been identified. Thus, the following sequence shows, the topmost left subterm to the innermost left subterm

```
Left Subterm: (((((a/c) * b) * c)/v(2))/v(1))
Left Subterm:  (((a/c) * b) * c)/v(2)
Left Subterm: ((a/c) * b) * c
```

```
Left Subterm:  (a/c) * b
Left Subterm:  (a/c)
```

The next step is to narrow these subterms from inner-to-outermost. Narrowing can only occur at a non-normalized subterm, so if the algorithm encounters a normalized subterm, it simply ignores it and continues onto the next one. The algorithm must also keep track of the number of delta rules used since before the topmost narrowing step: Once the number of invocations of the delta rule becomes equal to the bound for the delta rules, the algorithm can only narrow using the axioms of the IRQ, or the quandle TRS with the delta rules omitted. Consider now the sequence of possible subterms the algorithm will look at, and the successful narrowing and rewriting step it will take:

```
(a/c): Already normalized
(a/c) * b: Already normalized
((a/c) * b) * c: Already normalized
(((a/c) * b) * c)/v(2): Not normalized


Narrow (((a/c) * b) * c)/v(2) to a right cancellation rule
Unifier: [v(2)=c]
Narrowed subterm:  (((a/c) * b) * c)/c
Rewritten subterm: (a/c) * b
Rewitten overall term:
(((a/c) * b)/v(1))/v(2)
Current Unifier: [v(0)=v(1) * v(2),v(2)=c]
```

The subterm was replaced by the corresponding reduct to the redex it was narrowed, and the term has undergone now two total narrowing and rewriting sequences. The algorithm will now proceed recursively doing the sequence of the two previous steps (collecting the inner to outer most subterms, and then narrowing the first non-normalized subterm) until

the base case is hit. The base case is hit when all of the subterms of a given term have been normalized. As mentioned in chapter 5, a term is considered fully normalized when all of it's subterms are fully normalized. Recursively now working on the new, rewritten term, the inner to outermost subterms are now:

```
Innermost: a/c
Next Innermost:(a/c) * b
Next Innermost, Non-Normalized:((a/c) * b)/v(1)
Next Innermost, Non-Normalized(((a/c) * b)/v(1))/v(2)

First Non-Normalized Subterm Encountered:
((a/c) * b)/v(1)
Narrowing Step to Right Cancellation Redex:
Unifier: [v(1)=b]
Overall Unifier: [v(1)=b,v(2)=c,v(0)=star(v(1),v(2))]
Narrowed Subterm:
((a/c) * b)/b
Rewritten Subterm:
a/c
Rewitten Overall Term:
(a/c)*v(2)
```

The term has once again been transformed, and now it is clear that it is only one step away from full normalization. The next narrowing and rewriting step proceeds like so:

```
Innermost: a/c
Next Innermost:(a/c) * v(2)

First Non-Normalized Subterm Encountered:
(a/c) * v(2)
```

```
v(2) has previously been subbed out for c, so the term is:
(a/c) * c = Right Cancellation Redex
Term after rewriting:
a
Final Unifier:[v(1)=b,v(2)=c,v(0)=star(v(1),v(2))]
```

The unification algorithm has now run the course of narrowing and rewriting stage, as the only term, and subterm left is "a", which is fully normalized. The next stage is to check if the fully rewritten term is equivalent to the term that the original term was being unified with. In this case, the other term, after transformation, was simply "a". Therefore, the algorithm has found a successful unifier for the two terms. Now, the algorithm can be seen in all of it's steps like so:

```
Unification Algorithm:
        For some Term X and some other Term Y
        1. See if X can be made equivalent to Y through a single
        ↪  narrowing step. If not, proceed to the "non-simple" case
        2. Transform X and Y adding the "a * " prefix to each
        3. Cancel out X and Y by div'ing Y on both sides, leaving only a
        ↪  on one side
        4. Take a topmost narrowing step for X
        5. Take the truncated value of Log Base 2 of the number of
        ↪  operators and enforce the bound on possible delta rule
        ↪  invocations for the remainder of the algorithm
        6. Narrow first non-normalized innermost subterm and rewrite the
        ↪  term with this rewritten subterm
        7. Re-generate inner-to-outer subterms of new term
```

```
8. Repeat steps 6 and 7 until all subterms have been normalized
9. See if X = Y. If it does, there exists a unifier, if not, no
↪  unifier exists.
```

10

Prolog Implementation

The Prolog implementation for the unification algorithm will be a predicate of arity three, of which coordinate one and two will be the two terms that a unifier will be (or not be) found for, and coordinate three will be the unifier the algorithm will find after running through the algorithm. This predicate will be named "unification_algorithm". Consider the following query, in which the first two arguments are the two terms shown in the example of the previous chapter, and the third argument, "Unifier", is unified with the correct unifier:

```
?- unification_algorithm(divi(star(star(divi(a,c),b),c),v(0)),a,Unifier).
Unifier Found
Unifier = [v(0)=star(v(1), v(2)), v(1)=b, v(2)=c] ;
false
```

10.1 Checking For the "Simple Case" of Unification"

The unification algorithm will first execute steps one and steps two, which are to check if the "simple case" can solve the given problem, and if it does, do not backtrack and attempt the algorithmic approach. This is covered by the make_narrow_step_with_delta_rules predicate,

which takes either a narrowing step that includes the delta rule narrowing steps, or the "fully rewritten" narrowing cases in which the redex and reduct already are identical to a rewrite rule (these were covered in chapter 8). The predicate, like the remaining narrowing predicates that will be covered in this chapter, contain two additional arguments at the end. These are for the task of keeping track of the number of delta rule invocations in the narrowing and rewriting sequence. Coordinate four is the number of delta rule invocations before narrowing occurs, and coordinate five is the number of delta rule invocations after narrowing occurs. If the problem cannot be solved by the simple case, the program backtracks and attempts the algorithmic approach, in which the first steps are to "transform" the terms with "a * " predicate. So far, the predicate has the following structure:

```
unification_algorithm(Term,Term_2,Unifier):-
        (make_narrow_step_with_delta_rules(Term,Term_2,Unifier,0,New),!);
        transform_term(Term,Term1),
        transform_term(Term_2,Term2).
```

10.2 Reducing From an E-Unification Problem to an E-Matching Problem in Prolog

The predicate "transform_term" adds the "a *" prefix if it doesn't already exist in the current case, and the second argument is unified with the transformed version of the original term. The following examples show it's utility in querying and ability to avoid the "a *" prefix to terms in which the prefix already exists, even in deeply nested cases:

```
?-transform_term(star(b,c),Transformed_term).
  Transformed_term = star(a, star(b, c)).
```

```
?-transform_term(divi(star(star(divi(a,c),b),c),v(0)),New).
  New = divi(star(star(divi(a, c), b), c), v(0)).
```

The next step is to cancel the two terms to make it a matching problem in which all the variables have been shifted to one side, and thus the problem now requires only solving for a singular constant. This is done in the "cancel_terms" predicate, which takes in two terms, and returns their "matching" problem format. The following queries show how this works:

```
?- cancel_terms(star(a,divi(b,c)),star(a,v(0)),Term,Constant).
Term = divi(star(a, divi(b, c)), v(0)),
Constant = a .
```

The above predicate transforms the two terms such that one is simply the "a" constant and the other term now contains all the variables that existed in the original problem. In the code, these new terms are referred to as "Cancelled_1" and "Cancelled_2". Now that the transformed and cancelled terms have been derived, the next step is to take the topmost narrowing step, and subsequent subterm narrowing and rewriting steps.

10.3 Topmost and "Inner-To-Outer" Narrowing in Prolog

The unify_with_topmost_step, will take in the two terms that will be unified ("Cancelled_1" and "Cancelled_2) and find a unifier for them. This proceeds by first taking the topmost narrowing step, using the make_narrow_step_with_delta_rules predicate. The steps that make up the bulk of the unification algorithm, from taking the topmost narrowing step to the subterm narrowing, are encapsulated in a predicate called unify_with_topmost_step, which is written like so:

```
unify_with_topmost_step(Term1,Term2,Unifier):-
        make_narrow_step_with_delta_rules(Term1,Variant1,Sub,0,New),
        get_num_of_operators(Variant1,NumOps),
```

```
logbase2(NumOps,Delta_Rules_Bound),
non_var_positions(Variant1,NonVar),
reverse(NonVar,InnerToOuter),
narrow_all_subterms(InnerToOuter,Variant1,Term2,New,Delta_Rules_Bound,
Sub,Unifier).
```

It should be noted that the argument of coordinate four to the make_narrow_step_with_delta_rules is zero- this is because the delta rule invocation count begins at the stage of taking the topmost narrowing step after the transformation and cancellation steps. Thus, making coordinate four zero, which represents the delta rule invocation count before a narrowing step, shows that the count begins at zero before the narrowing step is taken at the topmost step. After taking this step, the argument of coordinate five will either be unified with zero or one- since a delta rule either is or is not taken, meaning the count after narrowing is either still zero, or a delta rule invocation did occur, and the count is now one. The next two predicate calls get the log base two value of the number of operators of the term after the topmost narrowing step is taken. This value is unified with the "Delta_Rules_Bound", which will be considered the bound on the delta rule invocations for the rest of the narrowing sequences. Next, a call to non_var_positions is made with the narrowed topmost term, known as "Variant_1". The predicate unifies the second argument with the following list:

```
?-non_var_positions(divi(star(star(divi(a,c),b),c),v(0)),Subterms).
Subterms = [divi(star(star(divi(a, c), b), c), v(0)), star(star(divi(a,
↪  c), b), c), star(divi(a, c), b),divi(a, c)].
```

The second argument is unified with a list of all the non-variable subterms of the term in coordinate one. It should be noted that the order of the list is structured in such a way so it proceeds from the overall topmost term to the nested, innermost subterm. Therefore, a call to the built-in reverse predicate will provide a new list with the reversed order of the

list that was created in non_var_positions. This will create a list in the order of the inner-to-outer format discussed in the previous chapter. Finally, a call to narrow_all_subterms is made, which will execute the inner-to-outer subterm narrowing, and will terminate when the base case of no more non-normalized subterms remaining is hit. When this base case is hit, the normalized term is compared to the other term, the singular constant, that was derived from the cancellation and transformation steps. If they are equivalent, then a successful unifier is found. Finally, once this step has been passed, all duplicates in the unifier list are removed with the built-in sort predicate. Therefore, the final structure of the unification predicate is laid out like so:

```
unification_algorithm(Term,Term_2,Unifier):-
        (make_narrow_step_with_delta_rules(Term,Term_2,Unifier,0,New),!);
        transform_term(Term,Term1),
        transform_term(Term_2,Term2),
        cancel_terms(Term1,Term2,Cancelled1,Cancelled2),
        unify_with_topmost_step(Cancelled1,Cancelled2,Unifier_),
        sort(Unifier_,Unifier).
```

"Unifier_" is the list that contains the duplicate substitutions, and therefore the argument in coordinate two is the final list of substitutions containing no duplicates. Some examples of this predicate's executions are such:

```
?- unification_algorithm(divi(star(star(divi(a,c),b),c),v(0)),a,Unifier).
Unifier Found
Unifier = [v(0)=star(v(1), v(2)), v(1)=b, v(2)=c] .
?- unification_algorithm(divi(v(0),v(1)),v(1),Unifier).
Unifier = [v(0)=v(1)].
?- unification_algorithm(star(star(a,b),v(0)),star(v(0),c),Unifier).
false.
```

```
?- unification_algorithm(star(divi(divi(star(star(divi(a,c),b),c),v(2)),v(1)),
v(2)),a,Unifier).
Unifier Found
Unifier = [v(1)=v(2), v(2)=star(v(3), v(4)), v(3)=b, v(4)=c] ;
Unifier Found
Unifier = [v(2)=v(1), v(2)=star(v(3), v(4)), v(3)=b, v(4)=c] ;
false.
```

According to sample data, unifiers are correctly found for those problems where one or more unifiers exist, and the predicate is considered false for two terms for which no unifier exist. Therefore, the algorithm may decide unification, as the algorithm appears to find all possible solutions for problems where a solution exists, and terminates and fails when no unifier exists. The issue of avoiding possible endless rewriting sequence due to the lack of the finite variant property has been solved if it can be shown this algorithm does decide unification over quandles.

11

Conclusion

11.1 Accomplishments

The paper successfully proved that the quandle TRS lacked the finite variant property through the usage of the Maude tool. A metavariable system was implemented in the Prolog unification program that allowed for more precise control over Prolog's unification abilities. Built on top of this metavariable system was a definition of the rewrite rules through narrowing relations that allowed for full narrowing abilities to be achieved in the unification program. Finally, a unification algorithm was derived that is terminating. The significance of this is that the lack of the finite variant property meant that, through narrowing, any rewrite sequence could be infinite in it's derivation length due to the delta rules. The unification algorithm is able to avoid these endless rewriting sequences through a combination of introducing a bound on the delta rules and by a transforming a term through introducing a constant as a prefix. This allows for a terminating unification algorithm since any term that cannot be unified with another will eventually reach a point it can no longer be narrowed or rewritten, and, if a unifier does exist, the term can find all such unifiers.

11.2 Further Work

The work laid out in this paper poses the problem of further solidifying a proof for the bound on the delta rule invocations that can occur throughout a given narrowing and rewriting sequence in the unification algorithm. The paper conjectures that the bound, that was found through the term structure:

$$nf(t_n)/v(0) = \mathrm{a}$$

where t_n begins with the constant a as well, is universal for all terms. This work is incomplete and further work should be done to verify this proof holds for all possible cases, not just the unification problem structure that the bound was derived through.

Bibliography

[1] Sam Nelson and Mohammed Elhamdadi, *Quandles: An Introduction to the algebra of Knots*, the Student Mathematical Library, vol. 74, American Mathematical Society, 2015.

[2] D Rolfsen, *Knots and Links*, Mathematical Lecture Series, vol. 7, Publish or Perish, 1976.

[3] S. Burris and H.P Sankappanavar, *A Course in Universal Algebra*, Springer Verlag, Berlin, 1981.

[4] AJ Tripathi, T.T. Tran Trang, T.T. Nguyen, and Robert McGrail, *A terminating and confluent term rewriting system for the pure equational theory of quandles* (2018), 157–163.

[5] David Joyce, *A classifying invariant of knots, the knot quandle*, Journal of Pure and Applied Algebra **23** (1982), 37-66.

[6] J.M. Belk and R.W. McGrail, *The Word Problem for Finitely Presented Quandles is Undecidable*, LNCS, vol. 9160, Springer, Heidelberg, Germany, 2015.

[7] H. Reidemeister, *Knoten und Gruppen*, Abh. Math. Sem., University of Hamburg, 1926.

[8] J. Wielemaker, *SWI-Prolog 5.6 Reference Manual*, 2006.

[9] P. Cromwell, *Knots and Links*, Cambridge University Press, 2004.

[10] Daniel and Johann Dougherty Patricia, *An improved general E-unification method*, Journal of Symbolic Computation **14** (1992), 303-320.

[11] Jeffrey Foster, Michael Hicks, and Neamtiu Iulian, *Understanding Source Code Evolution Using Abstract Syntax Tree Matching*, ACM (2005), 1-5.

[12] Daniel and Johann Dougherty Patricia, *An improved general E-unification method*, Journal of Symbolic Computation **14** (1992), 303-320.

[13] Jean and Snyder Gallier Wayne, *A General Complete E-Unification Procedure*, Lecture Notes in Computer Science (1987), 216-238.

[14] Franz and Nipkow Baader Tobias, *Term Rewriting and All That*, Cambridge University Press, 1998.

[15] Leon and Shapiro Sterling Ehud, *The Art of Prolog*, MIT Press, 1986.

[16] William and Mellish Clocksin Christopher, *Programming in Prolog*, Springer, 1998.

[17] Hans-Jurgen Burckert, *Matching-A Special Case of Unification?*, Journal of Symbolic Computation **8** (1989), 523-536.