Incorporating Equation Solving into Unification

Through Stratified Term Rewriting

by

Bing Zheng

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

APPROVED:

John W. Roach, Chairman

James P. Bixler

Edward A. Fox

December, 1988

Blacksburg, Virginia

Incorporating Equation Solving into Unification

Through Stratified Term Rewriting

by

Bing Zheng

John W. Roach, Chairman

Computer Science

(ABSTRACT)

This thesis studies equational theories incorporated into unification and describes STAR, a stratified term rewriting system that achieves a full integration. STAR is an advance over existing systems because it integrates an equational theory with unification at a lower, more fundamental level. Certain properties of STAR are proven including termination and confluence.

We also discuss the algorithmic complexity of the reduction algorithm, a vital component of STAR. We compare our system with narrowing and discuss the merits and drawbacks of each technique.

Since our system is an experimental integration of equation solving and unification, we are not concerned with the efficiency of the implementation. We do propose, however, some future improvements.

Acknowledgements

I wish to thank first of all my Lord for giving me the courage just in time to finish this work. I would like to thank Dr. Edward Fox for his many helpful comments and for being my inspiration. I also thank Dr. Pat Bixler for serving on my committee. Thanks go to Dr. Osman Balci for the use of the workstation without which this work would not have been possible. I want to thank Dr. John Dickey for always showing enthusiastism about ideas and Prolog. My heartfelt thanks go to many friends for their precious and steadfast friendship over the years. I am most grateful to my parents for their unconditional love, support, and patience. Finally, I owe my warmest thanks to John Roach, for his advice, love, and putting up with me; and may God bless him for the years to come.

Table of Contents

List of Illustrations

1. Introduction

This thesis describes a theory that incorporates equation solving into conventional unification. The computational model resulting from the theory is called stratified term rewriting. Our theory integrates functions into logic programming at a lower level than previous systems, and it demonstrates the power of stratified term rewriting systems. Lower level integration is more efficient and theoretically desirable than if we were to build separate logic and function evaluation systems that called each other. We illustrate the capabilities of our system, STAR, with an example and finally outline the remainder of the thesis.

1.1. The Goal

One of the most important and thoroughly researched modeling languages in computer science is first order logic. Resolution based logic programming, a major area of artificial intelligence research, is based on first order logic. Resolution is the computational model most frequently associated with first order logic. At the heart of resolution lies unification, a formalized syntactic pattern matching algorithm.

Unification has in recent years received increasing attention from researchers in the field of automated reasoning. Since unification is the heart of resolution, any improvement in the unification algorithm means a significant improvement in resolution itself. Efforts to improve unification, for example, include attempts to create a parallel unification algorithm, to speed up the occur-check, to create an algorithm for many-sorted logic, and to create an algorithm for combining function evaluation and logic programming.

Unification has a very limited, syntactic nature: two functions with a functor followed by an argument list unify only when the functors are identical and all their corresponding arguments unify. This becomes a serious problem in situations that involve numeric computation. As a simple example, if we try to unify X+3 with 10-1 or if we try to unify X+3 with 4+5 in a standard unification system, we will fail because "+" does not syntactically match "-" in the first case and 3 does not syntactically match 5 in the second — they are not identical functors or constants. As we can see, however, X=6 is the correct answer: we want to extend the unification algorithm to allow solutions to these kinds of problems.

Attempts to extend the power of unification have led to a search for ways to incorporate function evaluation into standard unification. Many efforts have been made to build function evaluation, or equation solving, into unification. Our method will provide an

arithmetic unification algorithm, one that treats both symbolic and numeric unification as the solution of simultaneous equations.

In this thesis, we will achieve arithmetic unification using term rewriting, a computational model that has recently emerged as an important field of study. Many people are developing techniques and applications for term rewriting systems. A major research direction concerns generating a canonical (i.e., noetherian - terminating - and confluent) term rewriting system from an equational theory and performing unification using the resulting system. There is a close relationship between unification and term rewriting that we shall fully develop.

This thesis is devoted to achieving arithmetic unification through term rewriting.

Unification reformulated as equation solving is much more powerful than standard or conventional unification. It eliminates some of the built-in deficiencies of standard unification such as its purely syntactic nature. Using (improved) term rewriting techniques to perform arithmetic unification sheds new light on the power and potential applications of term rewriting.

1.2. Arithmetic Unification and STAR

With the goal of achieving arithmetic unification, we have used an improved term rewriting system, called stratified term rewriting system (STAR), that achieves unification by solving simultaneous equations. For example, if STAR is requested to unify foo(X+3, W) with foo(6,4-W), it will unify the two expressions giving the answer W=2 and X=3. Therefore, STAR can not only evaluate but also solve equations in a system of numeric and symbolic computations. We are able to incorporate both the associativity and commutativity laws of the addition function into unification, although we do have to use an alternative set of rewrite rules rather than directly employing commutativity. That is, we do not have rewrite rules that look like: $x+y \rightarrow y+x$. Instead, we rewrite each numerical expression into a normal form. For example, both $5 \times z + 2 \times x + 4 \times y$ and $4 \times y + 2 \times x + 5 \times z$ are rewritten to $2 \times x + 4 \times y + 5 \times z$ and found equal by subtraction, instead of using commutativity and direct comparison. The rewrite rules to achieve this normal form are therefore more complicated. We will discuss them in greater detail in Chapter 4.

Our successful experimentation proves that stratified term rewriting is a very powerful computational model and that linear and some non-linear equation solving can be efficiently incorporated into standard unification.

Like every system, STAR has limitations. For example, STAR cannot solve all non-linear equations, including systems of equations involving transcendental functions (for example, exponential or triganometric functions). In fact, STAR mainly solves linear equations except for several very simple non-linear cases.

1.3. Example

We have run many examples using STAR. Here we discuss an outstanding example that illustrates what we have achieved. Consider our version of the factorial program, which is much simpler than the standard Prolog code:

```
factorial(0,1).
factorial(_n, _n&mul._x) :-
        factorial(_n-1, _x).
```

Now we can not only run a "normal" query, but also run it "backward". For example, the "forward" query "factorial(2,_V)" (meaning "what's the factorial for 2") will result in "_V = 2", while the "backward" one "factorial(_U,2)" (meaning "what's the number whose factorial is 2") results in "_U = 2".

1.4. Outline of the Thesis

In Chapter 2 we review previous work and systems developed in related areas, comparing their capabilities to STAR and explaining their defects. In Chapter 3 we analyse the technical difficulties of using term rewriting to incorporate solving systems of equations into unification and propose our solution. In Chapter 4 we present our arithmetic unification algorithm. Chapter 5 introduces stratified term rewriting and presents our system, STAR (Stratified Term rewriting for ARithmetic unification), for implementing arithmetic unification. Chapter 6 shows that STAR is terminating and confluent. Chapter 7 presents results illustrating the use of STAR. Chapter 8 sums up the thesis and gives a list of areas for future improvement.

2. Literature Review

This chapter reviews research work performed in related areas. We will first look at the standard unification and E-unification. Next we will study equational theories. Then we review term rewriting and narrowing. Afterward we look at unification treated as equation solving. Finally we give a brief overview of Bertrand and its use in our STAR.

2.1. Unification and E-Unification

The concept of resolution was first introduced by J.A. Robinson[Robi 65]. It introduced and used the concept of unification, which is the theme of this thesis. The unification algorithm that Robinson proposed always terminates and returns a unifier for the non-empty set A of terms to be unified. It is syntactic in the sense that all terms in A are (syntactically) identical after the unifier is applied to them.

There have been considerable research efforts to extend the standard unification algorithm. One of the extensions allows augmentation of unification with an equational theory and is called E-Unification. This extension was motivated by various computer science interests such as automatic theorem proving, software specification, term rewriting, and the combination of logic and functional programming. The most heavily studied equational theories include axioms for commutativity, associativity, idempotence, distributivity, and their combinations. Among the pioneering researchers in the field, M. Fay [Fay 79] gave a T-unification algorithm where T is an equational theory expressed using a technique known as "term rewriting." J.M. Hullot [Hull 80] studied the relationship between another technique called "narrowing" and unification and gave a new version of Fay's algorithm. He also gave a sufficient condition for the termination of the algorithm. P. Rety et al [ReKi 85] improved these algorithms even further. However, the term rewriting systems based on equational theories admitted in all of these algorithms are canonical, i.e., noetherian and confluent. This is a very heavy restriction on the equational theory since very often an equational theory does not have a corresponding canonical term rewriting system, such as when commutativity is involved, for example. Less restrictive term rewriting systems are certainly desired to describe equational theories.

J. You and P.A. Subrahmanyam [YoSu 86] were able to develop an E-unification algorithm that does not rely on the termination of the term rewriting system involved. Their system only has to be closed, left linear, and non-repetitive (while certain input terms do not require the non-repetition property). See [YoSu 86] for definitions and details.

2.2. *Equational Theories*

Typical equational theories contain the following elements: a set of (numerical) operators, a domain (usually a set of numbers) on which the operations are to be applied, and a set of equations that define the operational semantics of the equational theory (or, the scope of calculations). Following is an example of an equational theory:

1. $X+Y=Y+X$
2. $(X+Y)+Z=X+(Y+Z)$
3. $X+0=X$
4. $X+(-X)=0$

where "+" and "-" are the only operators in the theory, and the set of rational numbers is the domain.

There are two main ways in which researchers have incorporated equation solving into unification: equational logic and computational logic. Equational logic is in fact function evaluation without any genuine inference capability: it cannot make any inference until all parameters are ground. Computational logic, on the other hand, is a deduction engine and works at the same level as unification - i.e., it is a full inference engine for numeric computation (while resolution using unification is an inference engine for symbolic computation). With computational logic, we are able to perform numeric operations without having all the parameters be ground.

One of the major computational logic models is term rewriting. It is so important to the purpose of this thesis that we devote a separate section to it.

2.3. *Term Rewriting and Narrowing*

Term rewriting can be used as a computational model in areas such as automated theorem proving, programming language design and development, and databases. It is an operational interpretation of algebraic specifications from a logic point of view. For example, an algebraic specification in the form of an equation,
$x \times (y+z) = x \times y + x \times z$,
can be interpreted operationally as rewriting an expression of the form $x \times (y+z)$ into $x \times y + x \times z$. This is usually expressed as $x \times (y+z) \rightarrow x \times y + x \times z$ in term rewriting notation. That is, term rewriting defines exactly ***how*** to realise or implement the algebraic specification.

Typically, a term rewriting system is associated with an equational theory. In other words, term rewriting systems are often used to describe an equational theory. The set of equations describing an equational theory are typically made into a set of rewrite rules by simply changing all two-way equation relations into one-way rewrite relations. This change helps reduce the useless derivations and infinite loops caused by two-way equations by placing restrictions on the control. More specifically, we allow the derivation in one direction instead of both directions. This control is achieved by designing appropriate rewrite rules. A Knuth-Bendix completion algorithm [KnBe 70] is used to guarantee the confluence of this rule set by generating new rules utilizing critical pairs.

For the example equational theory we gave in Section 2.2 but without the commutativity axiom, the Knuth-Bendix procedure gives the following canonical term rewriting rules ([HuOp 80] and [Stic 81]):

1. $X + 0 \rightarrow X$
2. $X + -X \rightarrow 0$
3. $(X+Y)+Z \rightarrow X+(Y+Z)$
4. $0+X \rightarrow X$
5. $-X + X \rightarrow 0$
6. $-X + (X+Y) \rightarrow Y$
7. $-0 \rightarrow 0$
8. $-(-X) \rightarrow X$
9. $X+(-X+Y) \rightarrow Y$
10. $-(X+Y) \rightarrow (-Y) + (-X)$

There does not exist a canonical term rewriting system for the original equational theory.

The termination and confluency properties of term rewriting systems were heavily studied by researchers. Among them were: [Gorn 67], [Itur 67], [KnBe 70], [MaNe 70], [Gorn 73], [Lank 75a,b,77], [LiSn 77], [Plai 78a,b], [DeMa 79], [Lank 79], [KaLe 80], [Pett 81], [JoMu 84], [KaNa 85], [BaPl 85], [BaDe 86], [Rusi 87], and [Ders 87]. Basically they find some kind of ordering on terms or operators for the term rewriting system at hand, and prove termination under such orderings. Many theorems were proven along these lines, such as linearity, non-repetitiveness, etc. It turned out that there are different degrees of termination, such as weakly terminating, quasi terminating, and strongly terminating. Strongly terminating is also called finitely terminating, uniformly terminating, and noetherian. Termination is a necessary condition for confluency.

Term rewriting has an alternative form, an improvement, called "conditional term rewriting." Conditional term rewriting rules are similar to conditional instructions in programming languages and case analysis in proof methods. A conditional rewrite rule is a conventional rewrite rule with a condition attached to its end. For example, "$f(x) \rightarrow x$ if $x > 0$" is a conditional rewrite rule. It specifies that the rewrite of a term in the form "$f(x)$" into "x" takes place only when x is a positive number. The advantage of conditional rewrite rules is that some conflicts can be avoided. When more than one rule normally applies, for example, only the one with the appropriate condition would match in the corresponding conditional term rewriting system.

D. Brand [BrDa 78] and D.S. Lankford [Lank79] were the first to introduce conditional term rewriting systems. U. Pletat et al [PlEn 82], J. Bergstra et al [BeKl 82], and K. Drosten [Dros 83] have conducted research on rewrite rules' syntactic aspects such as linearity, while S.L. Remy [Remy 83] studied the confluence aspects. S. Kaplan [Kapl 84] extended the Knuth-Bendix procedure for conditional term rewriting systems.

Narrowing is basically the application of term rewriting to unifying a pair of terms. This usually happens in the context of an equational theory. In fact, narrowing is a key step in E-unification. For example, when we try to unify two terms, $X+3$ and $4+5$ under an equation theory including the addition function, we rewrite the terms first, into $X+1+1+1$ and $1+1+1+1+1+1+1+1+1$, respectively. This rewriting step is called narrowing. Then we apply the standard unification to the "narrowed" terms, and obtain the answer $X=1+1+1+1+1+1$, or $X=6$. Therefore, in some sense term rewriting is part of the narrowing process.

Narrowing was first introduced by J.R. Slagle [Slag 74] and then studied by M. Fay and J.M. Hullot [Fay 79] [Hull 80]. It is used in EQLOG [GoMe 85]. Narrowing's implementations include RITE [JoDe 86] and NARROWER [ReKi 85]. Recent works on narrowing include a study of basic narrowing techniques [Rety 87] and a new study of RITE [JoDe 89].

2.4. *Unification as Equation Solving*

Colmerauer has proposed a tree algebra to describe unification [Colm 82]. In this algebra, terms become trees, and infinite self referential structures are simply infinite trees - rational or irrational. Rational infinite trees can be represented by finite graphs. This makes it possible to accept rather than avoid infinite self referential terms. It thus becomes natural to represent binding or unifying relations in unification in the form of equations. Thus, Colmerauer developed the idea of treating unification as the solution of simultaneous (symbolic) equations.

From another point of view, treating unification as constraint satisfaction leads to the simultaneous equation solving idea.

Let us look at an example of how unification can be done in a tree algebra equation theory. Suppose we were to unify two terms, "foo(x,uux, y)" and "foo(y,x, uuuy)", where "u" is a unary operator. Colmerauer generates some equations for three sets:

set S that contains the equations x = uux and y = uuuy;
set T that is empty in the beginning;
set Z that contains the equation x = y.

After applying his reduction algorithm to these equation sets, Colmerauer obtains the resulting unifier set {x = uux, y = uuuy}.

2.5. *Bertrand*

Leler [Lele 88a] recently designed a constraint programming system called Bertrand. His original intention was to design a language that could be easily used to describe a constraint satisfaction system. He eventually created an "augmented term rewriting system" in which programmers can implement particular constraint satisfaction systems.

An augmented term rewriting system is one that adds the following features to conventional term rewriting systems:

- It has a way to bind values to atoms or variables using an "is" operator. During rewriting, when a variable (say X) is bound to some constant value (say 4), a binding would be generated in the subject expression in the form "X is 4" therefore binding X to 4. This new binding is treated as a new rewrite rule in the sense that ***all*** occurrences of X (globally) are rewritten to 4 by Bertrand.
- There are name spaces for variables. Each time a variable has a binding, the value to which the variable is bound to is stored in the corresponding name space for the variable. Also, there is a labeling system that keeps track of the rules and their variables.

- The user can define new data types and operators (such as "+") easily in Bertrand. Data types are treated as operators, and they form a hierarchy in which each data type may have at most one supertype and each supertype may have any (finite) number of subtypes.

Bertrand is a general purpose constraint satisfaction language constructor (like a compiler-compiler) that has augmented term rewriting rules as its statements. One of its major features is its capability to handle numerical equations - it can solve simultaneous equations as if they formed a constraint satisfaction system. Bertrand can also handle graphics and other problems by treating them as constraint satisfaction problems, and it has a better trade-off of efficiency and completeness than all other existing constraint satisfaction systems.

Compared with other (conventional) term rewriting systems, Bertrand is superior because it has features such as global substitution and data type hierarchy. Also unique to Bertrand is its uniform treatment of expressions and equations by treating the equal sign as a Boolean operator. Bertrand's expressive power allows the programmer to describe and solve simultaneous equations easily. In addition, since Bertrand allows the convenience of defining abstract data types and operators, it is relatively easy to add a new function to the system if it can be defined in terms of an equational theory. For example, we could add the multiplication function easily to the system by adding new data types and operators, and new rewrite rules to the system. No reorganization of the entire system is required to make such additions. Conditional rewriting systems may be able to simulate a type system added to rewriting by using type statements written as conditions (by analogy with the handling of types in the LOGIN language [AKNa 86]). If this technique could be made to work, however, it would run much more slowly than a rewrite system with type structure built directly into the language.

The following simple program illustrates some of Bertrand's features:

```
#type 'boolean
#primitive true 'boolean
#primitive false 'boolean
#op & left 100 'boolean
#op ~ prefix 200 'boolean
~true {false}
~false {true}
a & true {a}
true & a {a}
a & false {false}
false & a {false}
```

The first 3 lines define the data type hierarchy for the program. The following 2 lines define the operators of the program. Note that "left" and "prefix" here are associativity notations and "100" and "200" are precedence notations. The last 6 lines define the rewrite relations. Now if we want to run the query "true & ~ false & ~true", we append the following lines to the program:

```
main{true & ~false & ~true}.
```

Note that the operator associativity and precedence provide an implied grouping for the above expression. It is in fact grouped as follows: "(true & ~false) & ~true". Since no rule directly matches the expression, Bertrand rewrites the leftmost subexpression first,

which is "true & ~false". According to the rewrite rules, this subexpression is rewritten into "~false". Therefore the first rewrite step reduces the subject expression to: "~false & ~true" where the grouping is "(~false) & ~true". Since no rewrite rule matches directly, the next rewrite is applied to the subexpression "~false". So after the second rewrite the subject expression becomes "true & ~true". By the rewrite rule set, this is directly rewritten into "~ true" in the next step. And the next, final step rewrites "~ true" into "false" directly, which is the result of the query.

STAR is the first attempt to use Bertrand for arithmetic unification, although Leler claimed that it is a "common mistake" or "deficiency" to "try to use Bertrand like a logic programming language". But he "would love to see someone combine Bertrand with a logic programming language" [Lele 88b]. We are very proud that we have succeeded in using Bertrand in a logic programming context. There are some limitations, however, that prevent us from fully utilizing all of the advanced features of Bertrand, such as variable binding and string comparison - the "is" operator does not work for symbolic (non-numeric) values and Bertrand is unable to compare two strings alphabetically. Such limitations greatly reduce the efficiency of STAR. Nevertheless, STAR is still an innovative arithmetic unification theoretically.

3. Problem Analysis

In this chapter, we discuss in more technical detail the background of term rewriting systems and the difficulty of applying them to arithmetic unification; we also propose our solution to the problem.

First we show the undecidability and therefore the undesirability of a full function evaluation built into unification. Next we set up a restriction on the functions that we shall allow. Finally we propose our stratified term rewriting solution.

3.1. Complete Function Evaluation

Normal unification works by syntactic matching; functions are never evaluated. Our goal in this thesis is to incorporate a function evaluation capability into unification. Here by "incorporating" function evaluation or equation solving into unification, we mean the following procedure. When a set of equations between unifying pairs is set up to derive a unifier by the unification algorithm (as in [Colm 84]), we add to these equations the set of equations that make up the equational theory we are incorporating. For example, if our equational theory includes the addition function, and we are trying to unify the terms foo(X+3, 1) and foo(Y+4, Y-3), we would first derive the set of equations {foo = foo, X+3 = Y+4, 1 = Y-3}, and add to it the equations that specify properties of addition:

$$X + 0 = X$$
$$X + -X = 0$$
$$(X + Y) + Z = X + (Y + Z)$$
$$X + Y = Y + X$$

Given this set of equations, a unification algorithm that incorporates addition function would give the answer set of equations {X=5, Y=4}.

The full theory of function evaluation was discovered by Church ([Chur 41]); he called his theory the λ calculus. Unfortunately, λ calculus is equivalent to third order logic, and Huet has proven that unification in third order logic is undecidable ([Huet 73]). So unification incorporating λ calculus must be undecidable, too. This immediately directs our attention away from incorporating the full λ calculus into unification because we want an algorithmic unification system.

3.2. Lowering Expectations

Although we cannot build a complete function evaluation capability into unification, we can still build in some useful functions. This naturally creates a tradeoff between decidability and completeness. A complete function evaluation capability (in the extreme, using λ calculus) results in undecidability; decidability, therefore, requires a certain degree of incompleteness. Since we cannot have full completeness, we must choose a subset of evaluable functions to incorporate into unification. This choice must allow unification to be decidable while at the same time achieving as much completeness as possible.

The idea of restricting the class of functions was inspired by Ernest Davis (see [Davi 87]). He classifies the evaluable functions into four kinds: transcendental (e.g., exponential, sin, cos, etc.), polynomial, linear, and inequalities. We shall work only with equalities (i.e., non-inequalities) in this thesis. Solving simultaneous equations with polynomials requires exponential time (see [Davi 87]). Therefore we choose to include in our arithmetic unification algorithm only linear functions.

Note that even though we do not evaluate functions with higher complexity than linear, we do keep equations involving non-linear functions as constraints on future unification and perform non-linear function evaluation when all the necessary variable bindings are complete. For example, we keep the non-linear equation

$$3 \times X \times X \times X + 4 \times Y - 4 = 0$$

until X is bound to 2, which yields "Y = -5". We do not solve this equation without X being bound to a constant term.

3.3. Stratified Term Rewriting as a Solution

Now that we have decided to incorporate linear equation solving into unification, we need a tool in addition to an equational theory to achieve it.

In the long struggle to incorporate function evaluation into unification, two approaches stand out as the most promising: equational logic and computational logic. Many people have worked on the computational logic approach, and we chose this approach, too, because we think it has the most potential for success. Of course, another reason is to show that a powerful term rewriting system like Bertrand can be used to implement logic programming languages.

An important aspect of computational logic is the theory of term rewriting. The idea is to generate a canonical term rewriting system from the equational theory that we wish to incorporate in unification ([HuOp 80]). One way to do this is to first change the "="s in the equational theory to "→"s (thus obtaining a set of rewrite rules), then run the set of rewrite rules through the Knuth-Bendix completion algorithm to form a canonical term rewriting system.

A corresponding canonical term rewriting system, however, does not always exist since an equation stating commutativity when written as a rewrite rule sends rewrite systems into an infinite loop. We can therefore see that standard term rewriting is not enough

for our goal: incorporating a complete equational theory into unification without sacrificing the power of standard unification.

Let's look at the problem another way. If we could find an alternative to the commutativity rule, then we might be able to create a canonical term rewriting system. Perhaps we could change the commutativity rule syntactically while maintaining its semantics. This thought gives rise to the following idea: whenever two terms normally commute, our new rule(s) shall be directional, i.e., commutation of these two terms can occur in only one direction. Intuitively, this means that when we have a (sub)expression in the form X + Y, we may or may not be able to apply our new commutativity rule(s) to it; and in case we may, the result of the rewrite rule should prevent its being rewritten to X + Y again, directly or indirectly.

We do not rule out the possibility that several new rules may be needed to replace the rule "X + Y → Y + X". As a matter of fact, we may even make the new rules specific to our needs (i.e., not generic).

The idea of creating rules that may only fit our particular purpose, e.g., to solve linear equations, leads to the following thought: we can make up a convention so that all numeric expressions have a **standard form**, and our rewrite rules rewrite numeric expressions in such a fashion that they are only rewritten toward this **standard form**. This should help us solve our problem with the commutativity rule.

One obvious standard form for an arbitrary numeric expression groups and arranges all of its linear terms in alphabetical order and puts them to the left of the expression.

For example, the standard form of the non-linear expression

$$X^3 \times 4 + Y \times 5 - 6 \times W$$

could be rewritten to

$$-6 \times W + 5 \times Y + 4 \times X^3.$$

This recognition of data forms (e.g., recognizing whether something is a linear term) naturally calls for typing the data and ordering of the operators by associativity and precedence. That is, in such a system rewrite rules need to match only objects that have the proper types. This typing of data apparently reduces the number of rules all of which would otherwise match one object. The operator ordering, on the other hand, also reduces the overlapping of rules (two rules are said to overlap if they can both match some common object).

For example, suppose the following rules are present in a term rewriting system:

X + Y + Z → X + (Y + Z)
X + Y → (X) + Y

The object "a + b + c" could now be rewritten to any of the following forms:

a + (b + c)
a + ((b) + c)
(a) + (b + c)
(a + b) + c
.
.
.

where a, b, and c are constants.

But if we add type constraints to the rewrite rules, we would have the following new rules:

$$X'constant + Y'constant + Z'constant \rightarrow X + (Y + Z)$$
$$X + Y \rightarrow (X) + Y.$$

The same object will now only match the first rule and return "a + (b + c)" (assuming that a, b, and c are declared as constants). It would not match the second rule because the first rule is more "specific".

Term rewriting plus typing will hereafter be called "stratified term rewriting". The importance of typing was already shown in the example above. Typing eliminates collisions when matching and guarantees that the resulting system is canonical (i.e., terminating and confluent which we prove in Chapter 6.

3.4. Summary

In this chapter we discussed the goal of incorporating a full function evaluation capability into unification, its difficulties, and our decision to restrict the class of functions. We also described stratified term rewriting as a solution. The next two chapters shall present our solution in full detail.

4. Arithmetic Unification

In this chapter we review the concepts of normal unification, introduce an equational theory, and then present an arithmetic unification algorithm.

There are three sections in this chapter. Section 4.1 discusses normal unification. Section 4.2 presents an equational theory. Section 4.3 introduces arithmetic unification and presents an arithmetic unification algorithm.

4.1. Normal Unification

4.1.1. Terminology

First, we give some definitions that lead up to normal unification.

Defn. 4.1 — A **term** is defined inductively as follows:

A. A variable is a term.
B. A constant is a term.
C. If f is an n-ary function and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term.

Terms without variables are called **ground terms.**

Defn. 4.2 — If p is an n-ary predicate and $t_1, ..., t_n$ are terms, then $p(t_1, ..., t_n)$ is an **atom.** Specifically, atoms without variables are called **ground atoms.**

Defn. 4.3 — A **literal** is an atom or the negation of an atom, called a **positive literal** and a **negative literal** respectively.

Defn. 4.4 — A **simple expression** is either a term or an atom.

Defn. 4.5 — An **expression** is either a term, a literal, or a conjunction ($\wedge$) or disjunction ($\vee$) of literals.

Expressions without variables are **ground expressions.**

Defn. 4.6 — A **substitution** is a function that matches a finite set of distinct variables to a set of terms. We write a substitution in the form $\{ x_1 = t_1, ..., x_n = t_n \}$ where x_i is

matched to t_i, i = 1 , ... , n. and each t_i is distinct from v_i . The matching of variable v to term t is called a **binding** for v, and v is said to be **bound to** t.

When speaking of a substitution for a simple expression E, we mean a substitution for the set of variables in E.

For example, θ = {X=3, Y=4} is a substitution for the term x+y; (x+y) • θ yields the expression 3+4. Traditional unification algorithms cannot perform arithmetic, so 3+4 is left unevaluated.

The following is a key concept in the context of unification.

Defn. 4.7 — Let S be a finite set of simple expressions. A substitution θ is called a **unifier** for S if S • θ is a singleton (i.e., a set with one element). A unifier θ for S is called a **most general unifier (mgu)** for S, if for each unifier σ of S there exists a substitution γ such that $\sigma = \theta \bullet \gamma$

For example, for S = {f(x,a), y, f(b, z)}, θ = {x=b, y=f(b,a),z=a} is a unifier, as S • θ = {f(b,a)}. In this particular case θ is also the most general unifier.

Defn. 4.8 — The process of finding a unifier θ for a set of simple expressions S is called **unification.**

Although unification is about one pair of simple expressions, in reality we find it very common to unify a set of simple expression pairs simultaneously. For the sake of representation, we denote each pair in the form of a special equation, hereafter called the **unifying equation,** s_i UNIFY t_i, where s_i and t_i are simple expressions to be unified. For simplicity, we sometimes call unifying equations simply "equations" in the rest of the thesis. We represent unification differently than usual because we are going to deal with numeric equation solving in our arithmetic unification introduced below. The idea of equation notation came from Colmerauer (see [Colmerauer]). The theme of Chapter 5 will be solving (different kinds of) simultaneous equations.

Defn. 4.9 — **Applying a unifier to a set of unifying equations** means applying the unifier to each variable occurring in any unifying equation in the set.

For example, given the set of equations S = {f(X,4) UNIFY f(W,Y),V UNIFY 3} and the unifier U = {W=4,X=3}, S • U = S′ = {f(3,4) UNIFY f(4, Y), V UNIFY 3}.

4.1.2. Algorithm

We present here a version of the unification algorithm. The novel feature of this presentation is the treatment of normal unification as a solution of simultaneous equations.

Unification Algorithm:

Given initial sets S and U, where S is a set of unifying equations, and U is empty (the unifier-to-be), the following procedure computes the unifier U, leaving S empty.

1. If S is not empty, choose any equation

   s UNIFY t

in S and remove it from S. Apply the following steps to this equation.

2. If s and t are the same variable, go to step 1.

3. If either s or t is a variable that does not occur in the other, then add s = t or t = s to U, making sure a variable is on the left hand side, and modify S by applying U to it, then go to step 1; otherwise (either s or t is a variable that occurs in the other), fail.

4. If s and t are the same constant, then go to step 1; if s and t are different constants, then fail.

5. If s and t are of the form $f(s_1, ..., s_n)$ and $f(t_1, ..., t_n)$, $n \geq 1$, then add the following new unifying equations to S: s_i UNIFY t_i, i = 1, ..., n and go to step 1.

6. If s and t are of the form $f(s_1, ..., s_n)$ and $g(t_1, ..., t_m)$ where m, $n \geq 1$ and either $m \neq n$ or $f \neq g$, then fail.

4.1.3. Example

As an example, let S be the set { f (X, a, 4) UNIFY f (b, Y, Z), c UNIFY W } where capital letters represent variables. After unification, we get the unifier U = {X=b, Y=a, Z=4, W=c}.

The following is an account of what happens when we apply the algorithm to S.

Initially, S is not empty, so we choose "f(X,a,4) UNIFY f(b,Y,Z)" and remove it from S which then becomes {c UNIFY W}. Apply step 5 to the equation, and add to S "X UNIFY b", "a UNIFY Y", and "4 UNIFY Z", making S {X UNIFY b, a UNIFY Y, 4 UNIFY Z, c UNIFY W}.

Now we have finished one round, and start the second by choosing and removing "X UNIFY b" from S. Apply step 3 and add X=b to U. Applying U to S, S remains unchanged because X does not occur in any equation in S.

The third round starts with "a UNIFY Y" being removed from S. Apply step 3 and add Y=a to U. Applying U to S, S remains unchanged.

Start the fourth round by removing "4 UNIFY Z" from S. Apply step 3 and add Z=4 to U. Apply U to S which remains unchanged.

Start the fifth round by removing "c UNIFY W" from S. Apply step 3 and add W=c to U. Apply U to S which remains unchanged.

S is now empty, so the unification terminates successfully, leaving S empty and U equal to {X=b, Y=a, Z=4, W=c}.

4.2. Equational Theory

Notice that the unification algorithm we presented could not find a unifier for the set {X + 3 UNIFY Y + 4}, because $3 \neq 4$ although f = g ("+" in this case), thus failing the unification. Now, we want the unification algorithm to be more powerful, to be able to accept the concept of an equational theory and get the correct answers ({X=4, Y=3} in this case).

In order to achieve this, we shall first define the equational theory that we want to incorporate into normal unification. Our equational theory handles all of the following operators when their operands are constants:

$$+, -, \times, /, \wedge, \wedge\wedge, -, \sin, \cos, \tan, \text{atan}, \text{floor}, \text{round}$$

where $\wedge$ is the power operator (a $\wedge$ b means a^b) and $\wedge\wedge$ signifies scientific notation (a $\wedge\wedge$ b means a $\times$ 10^b). The two $-$'s are subtraction and unary minus, respectively. The functions sin, cos, tan, and atan are the usual trigonometric functions. The floor and round functions are unary operators where floor(x) is the greatest integer equal to or less than x, and round(x) is the integer closest to x.

We handle some non-constant and non-linear equations as well, as described by the following simplification equations.

- $X + 0 = 0 + X = X$
- $X - Y = X + (-1 \times Y)$
- $X \times 0 = 0 \times X = 0$
- $1 \times X = X \times 1 = X$
- $X / a = 1/a \times X$
- $-X = -1 \times X$
- $-a = 0 - a$
- $X + Y = Y + X$
- $X \times Y = Y \times X$
- $(X + Y) + Z = X + (Y + Z) = X + Y + Z$
- $(X \times Y) \times Z = X \times (Y \times Z) = X \times Y \times Z$
- $X \times (Y + Z) = X \times Y + X \times Z$
- $X \wedge 0 = 1$
- $X \wedge 1 = X$
- $a \wedge\wedge b = a \times 10 \wedge b$

We are assuming the linearity of all the variables occurring above, i.e., we assume that all capitalized letters represent only linear arithmetic expressions as defined below, and we also assume that the lower case letters represent ground expressions.

Defn. 4.10 —A **numeric (or arithmetic) expression** defined by our equational theory is any combination of variables and real numbers operated on by any combination of numeric operators in the equational theory.

Defn. 4.11 — A **linear (arithmetic or numeric) expression** is defined inductively as follows.

1. A single variable is a linear expression called a **variablized (linear) expression.**
2. A real number is a linear expression.
3. Real numbers operated on by any combination of arithmetic operations in our equational theory form a **ground (or constant) linear (or numeric) expression.**
4. Two (variablized) linear expressions connected by + or — form a new (variablized) linear expression.
5. A variablized linear expression connected by "×" with a ground linear expression (or constant (numeric) expression) forms a new variablized linear expression.
6. A variablized linear expression connected by "/" with a non-zero, ground linear expression (with the ground expression being the divider) forms a new variablized linear expression.

Defn. 4.12 — A **non-linear expression** is a numeric expression that is not linear.

Most non-linear expressions are not handled in our theory mainly due to poor efficiency and the general unsolvability of equations involving transcendental operators such as sine, cosine, etc., as explained in [Davis].

Defn. 4.13 — A **numeric context** is defined as follows:

1. Any numeric constant is in a numeric context.
2. For any legal term of the form $a(t_1, ..., t_n)$ where a is an arithmetic operator in the equational theory and $n \geq 1$, the term itself, $a(t_1, ..., t_n)$, and every t_i ($i = 1, ..., n$) is in a numeric context.
3. For any equation $s = t$, if one of s or t is in a numeric context, then the other is too.

For example, variable X in the term $4 + X$ is in a numeric context, according to the second rule in the definition above.

As another example, the expression $4 \times f(X, Y) + 3$ is illegal because the subterm f(X, Y) is not a defined numeric expression.

4.3. Arithmetic Unification

Now that we have both normal unification and an equational theory, we shall expand (or modify) the normal unification algorithm into a more powerful arithmetic unification algorithm.

4.3.1. Overview

Note that we cannot solve non-linear equations with unbound variables, so we are likely to have some unsolved non-linear equations along with the unbound variables involved by the end of a unification. These equations can be resolved only when enough variables become bound to ground numeric expressions. Therefore in the following algorithm we need a number of different sets:

- **U(nify)** which contains "normal" unifying equations to be solved, each in the form "s UNIFY t"; this is form of the input;
- **G(round)** which contains bindings of variables to ground terms (numeric or non-numeric);
- **F(unction)** which contains equations of the form $v = f(t_1, ..., t_n)$ where $n \geq 1$, v is a variable, and f is not an arithmetic operator defined in our equational theory;

  Equations of the form "variable = variable" belong in this set, too, since before a variable is bound, it has the potential to be bound to either symbolic or numeric values. If we put "variable = variable" in **L(inear)** as defined below, we eliminate the possibility that these two variables involved may take symbolic values.
- **L(inear)** which contains equations of the form $v = a(t_1, ..., t_n)$ where $n \geq 1$, v is a variable, and a is an arithmetic operator defined in the equational theory;
- **N(onlinear)** which contains equations of the form $q = 0$ where q is a non-linear expression;
- **V(ariable)** which contains variables that have (or should have) numeric values. The necessity for V can be seen from the following example.

Suppose that during unification, variable x is found to occur in a numeric context. Now if we see "w = x", we would know immediately that w must have numeric values. This way the unification fails sooner should an equation like "w = f(a)", where f is not an arithmetic operator defined in our equational theory, come up later in the algorithm.

Apparently V is not part of the final most general unifier, but it can be viewed as a constraint on the unification algorithm.

Defn. 4.14 — If by the end of unification N is not empty, we say that termination of the unification is **partial**. Unification is said to be **complete** if N is empty upon termination. (Note that partial completion occurs when the set of simultaneous equations is underdetermined and must be carried over to further unifications.)

Among all of these sets F, L, N, and G contain all the variable bindings that together make up the unifier. They have been separated because bindings change during unification, and because they have different **modification algorithms** (explained in detail below). We will present the **modification algorithms** after the unification algorithm. Basically, as the unification algorithm computes the most general unifier for the set of input equations, every time a new binding (in the form of an equation) is created, we need to modify some of the sets U, F, L, N, G, and V to be sure that the binding is substituted appropriately throughout the whole list of equations.) Note here that we start with a possibly non-empty set of unifiers (carried over from previous unifications in the context of a logic program with arithmetic involved, for example).

Before giving the formal algorithm, we will describe the algorithm informally.

We can get a sense of how the arithmetic unification algorithm works as follows. We will generate variable bindings just as in normal unification, but when arithmetic functions are encountered, we allow **reduction** (which is in a sense function evaluation) instead of one-to-one syntactic argument matching. For example, if we have $3+X$ UNIFY $4+Y$, we generate the equation $3+X-(4+Y)=0$ and reduce it to $X=Y+1$, which is a binding for X. As another example, we reduce

$$"5 \times Z + X \times (Y \times 4 + Z / 5) + Z \times 4 = 0"$$

to

$$"9 \times Z + 4 \times X \times Y + 0.2 \times X \times Z = 0"$$

which does not yield an immediate variable binding because it is a non-linear equation, and we have no way of reducing either "$4 \times X \times Y$" or "$0.2 \times X \times Z$". We also cannot reduce the equation $3 \times X \wedge 2 - Y = 0$. In this case we get an unresolved equation and some unresolved variables (X and Y here). Later on in the unification process, however, we may determine that $X=4$, and with appropriate substitutions, the non-linear equation has become linear. This is why we must modify N, the set of non-linear equations. For similar reasons, L is a separate set, and its members are "candidates" for later inclusion in G, the set of ground bindings of variables. Similarly, bindings of variables to general functions (represented by equations in F) may become ground, too. But the ways to work with these bindings are so different that we decided to distinguish them by means of categorization — i.e., by sets.

During unification, we check the sets U, G, L, F, N, and V each time a new equation is generated. Other than the fact that we have more types of bindings (signified by those sets) and that equations may move around between sets when new bindings occur, the following algorithm is based on the same principles as the one in Section 4.1.

The **arithmetic unification algorithm** described below calls for **modification algorithms for sets V, U, F, L, and N**. It also calls for a **reduction algorithm**. All of these "accessary algorithms" will be presented after the main algorithm.

4.3.2. Algorithm

Given the initial sets U, G, F, L, N, and V each as explained above, the following procedure computes G, F, L, N, and V until either U is empty or failure occurs.

1. If U and N are both empty, terminate successfully.
If U is empty but N is not, terminate partially.
If U is not empty, choose any equation in U: s UNIFY t, and remove it from U.

2. If s and t are the same constant, then go to step 1.
If s and t are different constants, then fail.

3. If s and t are the same variable, then go to step 1.
If s and t are different variables, and none of them was in V before, then add s = t or t = s (whichever is alphabetically smaller is on the right hand side) to F and call the modification algorithms for U and F — {s = t (or t = s)} respectively with argument s = t (or t = s); if at least one of them was in V before, then add s = t or t = s to L and call the modification algorithms for U, L — {s = t (or t = s)}, F, and N. Go to step 1. (This basically substitutes all occurrences of s in U, V, F, L, and N by t, and check the consistency of the resulting equations.)

4. If one of s or t is a variable, say s for argument's sake, and s occurs in t, then fail. If s does not occur in t, and t is:

A. a constant, then add s = t to G, call the **modification algorithms** for V, U, F, L, and N respectively with s = t, and go to step 1. (This substitutes all occurrences of s in the input and current non-ground variable bindings by t. It also puts s(t) into V should t(s) already be there.)
B. a function in the form $f(t_1, ..., t_n)$ where n ≥ 1, and f is not an arithmetic operator in the equational theory, then add $s = f(t_1, ..., t_n)$ to F, call **modification algorithms** for U, V, F — $\{s = f(t_1, ..., t_n)\}$, L, and N respectively with $s = f(t_1, ..., t_n)$, and go to step 1. (This substitutes all occurrences of s in the input and current non-ground variable bindings by t, and puts s(t) into V should t(s) already be there.)
C. a function in the form $a(t_1, ..., t_n)$ where n ≥ 1 and a is an arithmetic operator in the equational theory, then call the **modification algorithm** for V with $s = a(t_1, ..., t_n)$ and call the **reduction algorithm** with $s - a(t_1, ..., t_n) = 0$. If the result is:

i. false, then fail.
ii. true, then go to step 1.
iii. v = a, where v is a variable and a is a (numeric) constant, then add v = a to G, call the **modification algorithm** for U, F, L, and N with v = a, and go to step 1.
iv. v = t′ where v is a variable and t′ is a variablized linear expression then add v = t′ to L, call the **modification algorithms** for U, F, L — {v = t′}, and N with v = t′, and go to step 1.

For example, suppose s is the variable X and t the expression 3 + 4 × Y, then we first call the modification algorithm for V and make sure that both X and Y are in V, then call the reduction algorithm with X-(3 + 4 × Y) = 0, and get

$X = 4 \times Y + 3$ in return. This is a linear binding of X, so add it to L, and modify U, F, L-$\{X = 4 \times Y + 3\}$, and N respectively.
v. $q = 0$ where q is a non-linear expression, then add $q = 0$ to N and go to step 1.

For example, suppose s is X and t is $4 \times X \times Y - 5 + W \times X$, then we want W, X, and Y to be in V (therefore we call the modification algorithm for V), then reduce the equation $X - (4 \times X \times Y - 5 + W \times X) = 0$ by means of the reduction algorithm, and get the resulting non-linear equation $X - 5 + 4 \times X \times Y + W \times X = 0$ which we add to N. There is no new variable binding so we do not have to modify any set this time.

5. If one of s or t is a non-numeric constant, and the other is any function (whether or not in the equational theory) $f(r_1, ..., r_n)$ where $n \geq 1$, then fail.
If one of s or t is a numeric constant and the other is a function $f(r_1, ..., r_n)$ where $n \geq 1$ and f is not an arithmetic operator in the equational theory, then fail.

6. If s and t are of the form $f(s_1, ..., s_m)$ and $g(t_1, ..., t_n)$ where m, $n \geq 0$ but such that s and t do not fit any of the previous cases, and

A. neither f nor g is an arithmetic operator in the equational theory, and either $m \neq n$ or $f \neq g$, then fail.
B. neither f nor g is an arithmetic operator in the equational theory, and $m = n$ and $f = g$, then add s_i UNIFY t_i, $i = 1, ..., n$, to U; go to step 1.
C. one of f and g is an arithmetic operator in the equational theory, and the other is not, then fail.
D. one of f or g is an arithmetic operator in the equational theory, say f, and the other term (t) is an arithmetic expression (i.e., either g is an arithmetic function defined in the equational theory or $n = 0$ in which case t is a numeric constant), then call the **modification algorithm** for V with $s - t = 0$ and F, and call the **reduction algorithm** for $s - t = 0$. If the result is:
i. false, then fail.
ii. true, then go to step 1.
iii. $v = a$ where v is a variable and a is a constant, then add $v = a$ to G, call the **modification algorithms** for U, F, L, and N with $v = a$, and go to step 1.
iv. $v = t'$ where v is a variable and t' is a variablized linear expression then add $v = t'$ to L, call the **modification algorithms** for U, F, $L - \{v = t'\}$, and N with $v = t'$, and go to step 1.
v. $q = 0$ where q is a non-linear expression, then add $q = 0$ to N and go to step 1.

The following are the **modification algorithms** for V, U, F, L, and N respectively.

Modification Algorithm for V

Given an equation of the form $s = t$ with which to modify V, if s and t are both variables and one of them is in V, then add the other to V if it is not there already. If s and t are not both variables, then add to V all variables occurring in s or t that are in a numeric context (see Defn. 4.13) if they are not yet in V. If one of s and t is a variable and was in V, say s, and t is a non-numeric constant or function that is not defined in our equational theory, then fail.

To modify V with F, simply modify V with each equation of the form "var1 = var2" in F.

Modification Algorithm for U

Given an equation of the form v = t with which to modify U, where v is a variable and t a term, substitute all occurrences of v in U by t.

Modification Algorithm for F

Given an equation of the form v = t with which to modify F, where v is a variable and t a term, substitute all occurrences of v in F by t. If the right hand side of an equation becomes ground after substitution, remove this equation from F and add it to U, changing "=" to "UNIFY".

For example, F = {X = f(Y,W), Z = g(a,V)} becomes {X = f(Y,W)} if modified by "V = 4", and "Z UNIFY g(a,4)" is added to U.

A more efficient way of doing this would be to add the ground binding (Z = (a,4) in this case) directly to G instead of to U, but this new addition to G would cause recursive modifications to U, F, L, N, and V, which would be undesirable. This can be shown as follows. By having a new equation in G, we modify U, L, F, and N respectively. There is the possibility that some non-ground variable binding becomes ground and that some non-linear equation yields a linear binding for some variable. In this case, each of the newly changed bindings would cause recursive modification before the original modification is done, which creates undesirable complications.

Modification Algorithm for L

Given an equation of the form v = t with which to modify L where v is a variable and t a term, look at each equation x = g in L, where x is a variable and g a linear expression.

- If v and x are the same variable, then add "g UNIFY t" to U.
- If v occurs in g, then fail if t is a non-numeric expression; otherwise substitute all occurrences of v in g by t, getting x = g'. Remove x = g' from L and add "x UNIFY g'" into U.

Example
For example, suppose the equation (new binding) is V = W + 4. Suppose also that L has an equation V = X + 5. Now according to the first case, we add W + 4 UNIFY X + 5 to U and leave the other sets unchanged. Suppose now that the new binding is V = f(a) instead. We add f(a) UNIFY X + 5 to U instead.

As another example, suppose L has another equation U = V + 2. If the new binding is V = W + 4, then according to the second case, we substitute all occurrences of V in V + 2 by W + 4, and get the new equation U = W + 4 + 2. Remove this equation from L and add the corresponding "unsolved" equation "U UNIFY W + 4 + 2" into U. Now suppose that the new binding was V = f(a). According to the second rule, the modification algorithm fails because V has both a numeric value and a symbolic value.

Again, we could have reduced x — g' = 0 and made modifications to U, V, G, F, L, and N accordingly, but that would get into an undesirable recursion. So we put "x UNIFY g'" into U and leave the work for the next round of unification algorithm.

Modification Algorithm for N

Given an equation of the form v = t with which to modify N where v is a variable and t a term, look at each equation q = 0 in N.

- If v occurs in q, and t is a numeric expression, then substitute every occurrence of v in q by t getting q' = 0; if t is a numeric constant, remove q' = 0 from N and add "q' UNIFY 0" to U.

  Here we put "q' UNIFY 0" in U, because there is a possibility that q' can become linear. For example, 8 + X × Y × 5 — Y = 0 (where q' is 8 + X × Y × 5 — Y) would yield X = — 3.33 given Y = 3. Again we avoid recursion by putting "q' UNIFY 0" in U instead of reducing it directly.
- If v occurs in q and t is a non-numeric expression, then fail.

The following is the **reduction algorithm** we used in the unification algorithm.

Reduction Algorithm

Given a non-linear equation q = 0, where q is a non-linear expression, apply the following steps to q:

1. Flatten q so that no parentheses remain; call the resulting expression q'. We use the distribution law and our own rules to eliminate parentheses. One of our typical rules is: if there is a pattern (a) + b or a + (b), rewrite it into a + b. The distribution law is: X × (Y + Z) = X × Y + X × Z.

2. For each term in q', if it has a minus sign (be it unary or binary) in front of it, change the sign to plus and add the factor "-1" to the term. Evaluate the constant factors (coefficient for each term) and put it at the front of the term. "term" and "factor" here refer to the usual arithmetic terminology.

3. Move all linear terms (in the form c × V where c is a constant and V a single variable) to the left end of the expression.

4. Sort the linear terms in alphabetical order of the variables in the terms. Combine terms with the same variables where applicable.

5. If the resulting equation q'' is in the form a×X + b = 0 where a and b are constants and X is a single variable, then return X = -b/a (calculate the value of "-b/a" first); if q'' is in the form a×X + Y = 0 where a is a constant, X a single variable, and Y a variablized linear term, then return X = Y' where Y' is the result of putting the coefficient -1/a into Y; if q'' is in the form z = 0 where z is a non-linear expression, then return z = 0.

4.3.3. Complexity Analysis

Here is a complexity analysis of the reduction algorithm in 4.3.2.

Suppose there are n variables in the non-linear equation to be reduced (recall that an equation to be reduced must be in the form "non-linear = 0"), and m terms (those involved in distributivity operations; counted by occurrence, not by identity). For example, in the equation "(4 + X) * (Y-6) * Z = 0", there are three variables (X, Y, Z) and five terms (4, X, Y, 6, Z). And the equation "(3-U)*(2 + V)*(W + X) = 0" has six terms.

For m terms, the most expensive case for the flattening step would be one where terms occur in the form:

$$(t_1 + t_2) \times (t_3 + t_4) \ldots \times (t_{m-1} + t_m)$$

for m an even number, and in the form:

$$(t_1 + t_2) \times \ldots \times (t_{m-2} + t_{m-1}) \times t_m$$

for m an odd number. The complexity for this case is O($\frac{2^m}{2}$) (exponential), since there are $\frac{2^m}{2}$ multiplications to perform. The other extreme occurs in the form:

$$(t_1 + \ldots + \frac{t_m}{2}) \times (t_{\&half+1} + \ldots + t_m)$$

for m an even number and in the form

$$(t_1 + \ldots + \frac{t_{m-1}}{2}) \times (\frac{t_{m+1}}{2} + \ldots + t_m)$$

for m an odd number,

but the expense is much lower, since there are only $\frac{m}{2} \times \frac{m}{2}$, or $\frac{m^2}{4}$ multiplications. Therefore the complexity for the flattening step is O($\frac{2^m}{2}$). Note that the complexity here relies only on the number of terms, regardless of how many variables there are.

For the second step of the algorithm, making the signs uniform, the complexity is O($\frac{2^m}{2}$) also, because there would be $\frac{2^m}{2}$ terms to be examined.

For the third and fourth steps of the algorithm, there are $1+2+...+n+n+...+n$, i.e.,

$$\frac{2^m}{2} - n + \frac{n \times (n+1)}{2}$$

comparisons in the worst case, where all

$$\frac{2^m}{2}$$

terms from step 2 are linear terms and need to be sorted. Therefore the complexity for these two steps is O(

$$\frac{2^m}{2}$$

).

The complexity for the fifth step is a constant (0 or 1).

Therefore the complexity for the reduction algorithm is exponential, O(

$$\frac{2^m}{2}$$

), where m is the number of terms occurring in the original non-linear equation.

4.3.4. An Example

Unify foo(X, W*Z, X cons nil) with foo(W, W+8, 2 cons nil).

Initially we have:

U = {X UNIFY W, W*Z UNIFY W+8, X cons nil UNIFY 2 cons nil}

G = L = F = N = V = { }.

For the first round of the algorithm, we remove "X UNIFY W" from U. According to the algorithm (3), we add "W=X" to F, and modify U (which is now {W*Z UNIFY W+8, X cons nil UNIFY 2 cons nil}) and F-{W=X} (which is empty). U will become

{X*Z UNIFY X+8, X cons nil UNIFY 2 cons nil}.

This ends the first round.

For the second round of the algorithm, we remove "X*Z UNIFY X+8" from U first. According to 6D of the algorithm, we first modify V with X*Z - (X+8)=0 and F ({W=X}) to get a new V={W, X, Z}. Next we reduce X*Z - (X+8)=0 to get: -X -8 +X*Z =0. Add teh above equation to N.

For the third round of the algorithm, remove the last equation, "X cons nil UNIFY 2 cons nil", from U. According to 6B of the algorithm, add "X UNIFY 2" and "nil UNIFY nil" to U.

The fourth round of the algorithm starts with "X UNIFY 2" being removed from U. According to 4A of the algorithm, add "X=2" to G and modify V ({W, X, Z}), U({nil UNIFY nil}), F({W=X}), L({ }), and N({-X -8 + X*Z =0}) respectively with X=2. The result of these modifications is:

V = {W,X,Z}

U = {nil UNIFY nil}
F = { }; add "W UNIFY 2" to U
L = { }
N = { }; add "-2 -8 + 2*Z UNIFY 0" to U.

Therefore, U eventually becomes
{W UNIFY 2, -2 -8 + 2*Z UNIFY 0, nil UNIFY nil}.

The fifth round begins with "W UNIFY 2" being removed from U. According to 4A of the algorithm, add "W=2" to G and call the appropriate modfication algorithms. The sets involved are:

V = {W, X, Z}
U = {-2 -8 + 2*Z UNIFY 0, nil UNIFY nil}
F = L = N = { }

This finishes the round.

The sixth round starts with "-2 -8 + 2*Z UNIFY 0" being removed from U. According to 6D of the algorithm, we first modify V with "-2 -8 + 2*Z -0 =0" and F ({ }) which does not change V, then reduce the above equation to: Z = 5. According to 6Diii of the algorithm, add Z=5 to G, and modify U, F, L, N with Z=5 to get new sets:

U = {nil UNIFY nil}
F = L = N = { }.

For the seventh round, remove the last equation from U: "nil UNIFY nil". Since "nil" and "nil" are the same constant, this round is completed.

At the beginning of the eighth round, we find both U and N empty, and therefore terminate the unification process successfully, and the unification is complete (as opposed to partial).

4.3.5. Another Example

Unify foo(X*Y+4, Z) with foo(W,a).

Initially we have:

U = {X*Y+4 UNIFY W, Z UNIFY a},
G = L = F = N = V = { }.

For the first round of the algorithm, we remove "X*Y+4 UNIFY W" from U. According to 4C of the algorithm, we modify V to obtain a new V={W, X, Y}, and reduce the equation "W-(X*Y+4)=0" to "W-4-X*Y=0". According to 4Cv, add "W-4-X*Y=0" to N and finish the first round.

The second round starts with "Z UNIFY a" being removed from U. According to 4A of the algorithm, add "Z=a" to G and modify V, U, F, L, and N — but all of these sets remain unchanged.

At the beginning of the third round we find U empty but N not, so we terminate partially.

This example gives an intuition of what partial termination means.

5. Stratified Term Rewriting

In this chapter, we first review standard term rewriting concepts and then introduce the concept of data type hierarchy, followed by stratified term rewriting. Stratified term rewriting combines standard term rewriting with a data type hierarchy. Next we introduce the classification of equations, and finally, we introduce a stratified term rewriting system, STAR, for implementing the arithmetic unification algorithm described in Chapter 4.

5.1. (Standard) Term Rewriting

In this subsection we review the concepts of (standard) term rewriting.

Defn. 5.1 — A **rewrite rule** is an ordered pair of terms $< L, R >$ that we write, for clarity's sake, $L \rightarrow R$, where L is called the **head**, and R the **body** of the rewrite rule.

Hopefully, there is a similarity relation between the head and the body of a rewrite rule so that the rule is meaningful. For example, the rewrite rule

$$X \times (Y+Z) \rightarrow X \times Y + X \times Z$$

maintains the similarity relation (equality in this case) between its head and body, which makes the transformation meaningful.

Defn. 5.2 — Given a set of rewrite rules R and a simple expression E, the application of R that transforms E into another expression E′ is called **term rewriting**. E is called the **subject expression.**

Defn. 5.3 — Given a subject expression S and a rewrite rule $RULE_i$: $L_i \rightarrow R_i$, if there exists a substitution θ for L_i such that $L_i \bullet \theta = S$, then we say L_i **matches** S.

Defn. 5.4 — Given a subject expression E and a rewrite rule R_i, suppose the head of R_i matches a subexpression S of E with substitution θ . Apply θ to the body of R_i and use it to rewrite S into S′, thus transforming E into E′. This process is called **instantiating the rule** R_i.

Defn. 5.5 — A **(standard) term rewriting system** is a triple $< V, S, R >$ where V is the set of variables, S the set of subject expressions, and R the set of rewrite rules.

For example, the following is a simple term rewriting system $< \{X,Y\}, S, \{4+3 \rightarrow 7 , X+Y \rightarrow Y+X, X \times Y \rightarrow Y \times X\} >$ where S is the set of all possible linear expressions (an infinite set) as defined in Chapter 4.

5.2. *Data Type Hierarchy*

5.2.1. Introduction

Defn. 5.6 — A **data type** is a category of data.

For example, integer, number, string, variable, etc. are data types.

Defn. 5.7 — Given a domain that contains j data types, $j \geq 1$, a **data type hierarchy** can be built in the following iterative way. Let the original data types T_i^0 (i = 1,2,...,j) be of level 0. Let S^i be the set of all existing data types of level i ($S^0 = \{T_1^0, ..., T_j^0\}$). A new data type T_k^{i+1} of level i+1 can be built from a subset A of $S^0 U S^1 U ... U S^i$ by taking every data type in A to be a subtype. At least one subtype in A must be from S^i. T_k^{i+1} is called the **supertype** of each element of A. A data type can have at most one supertype.

For example, suppose that the original data types are: numbers(T_1^0), strings(T_2^0), and variables(T_3^0). Then $S^0 = \{ T_1^0, T_2^0, T_3^0\}$. If we take A to be $\{T_1^0, T_3^0\}$, then we have T_1^1 as a supertype with its subtypes being T_1^0 and T_3^0. T_1^1 contains numbers and variables that can only take numeric values.

5.2.2. Specific Data Type Hierarchy

Defn. 5.8 — In a unifying equation L UNIFY R, "UNIFY" is called an **equation sign.** In general, a unifying equation is in the form of L equation-sign R, where "equation-sign" can be any alphabetic-numeric string. STAR has 21 equation signs, which will be explained later in this chapter.

For example, "=" is the equation sign in the unifying equation

x = 4,

and "eq" is the equation sign in the unifying equation

y eq a.

The reason for so many equation signs will be explained later in this chapter.

Now that we have introduced the concept of a data type hierarchy, we are ready to generate a data type hierarchy for the equational theory described in Chapter 4.

Bertrand, the language we used to implement the arithmetic unification algorithm, has some primitive data types such as constant, literal, true, and false (see [Leler]). A "constant" is Bertrand's name for a number. A "literal" is Bertrand's name for a character string. The constants "true" and "false" make up a supertype of "boolean". Our data type hierarchy will be built using these primitive types.

Another aspect of the data type hierarchy is the associativity and precedence (optionally) defined for each data type. For example, the statements

#op + left 100
#op × left 200

specify an underlying ordering for the expression 3+4×5, namely 3+(4×5). When attempting to match this expression with rewrite rules at run time, 4×5 will be matched first if applicable, as opposed to 3+4. This ordering of the operators is very important for the termination and confluence of the algorithm, as we will see in Chapter 6.

The chart in Figure 1 fully describes the data type hierarchy of our stratified term rewriting algorithm for arithmetic unification.

According to this chart, our primitive types are: true, false, literal, and constant. The arithmetic operators are: +, − (binary), ×, /, ∧, ∧∧, round, floor, sin, cos, tan, atan, and unary −. Our operators for linear expressions are: + +, − −, and ××. Our equation signs include: eq, temp, numsim, numbar, numfin, gdsim, gdbar, gdfin, linsim, linbar, linfin, lissim, lisbar, lisfin, funsim, funbar, funfin, varsim, varbar, varfin, and = (the explanation for these equation signs will come later). The list construction operator is cons. The general function operator is func (a general function is a function which is not one of the arithmetic operations defined in the equational theory), and ";" is a metasymbol that separates the equations. The "," is used to separate arguments for general functions. Miscellaneous operators are else, then, [ ], ~, &, and |. We also have subroutines atom, freestr, ground, occur, onlyvar, pred, var, varonly, varc, and merge.

We will further explain these operators below. For the time being, just remember that names/signs followed by underlined words in parenthesis (associativity and precedence) are operators, and those that come alone are super- or sub- types.

Now consider the supertypes (nodes that have branches coming out) in the chart. A simple term, "sterm", refers either to a numeric constant (a number), a symbolic constant (an atom), or a variable. Notice that syntactically there is no distinction between a variable and an atom. We can find out whether a literal is a variable by calling the subroutine "var". A linear expression, "linear", was described in Chapter 4. Numeric expressions, "expression", were also described in Chapter 4, except that in case of a single variable it may later be bound to a non-numeric value even though it was considered a numeric expression originally.

We just explained the arithmetic side of the data structure for Figure 1. As for the symbolic side, we have lists (which may be considered to be the application of special functions for Figure 1) and general functions. The operators for them are cons and func respectively.

A "term" may be either a numeric expression, a list, or a general function. In practice, a general function may have lists, expressions, and other general functions as its arguments. A list may have general functions, expressions, and other lists as its arguments. Numeric expressions must have only numeric expressions as its components. This is not easily seen from the chart.

During the course of rewriting, illegal equations (e.g., 4=0) become "false", while identity equations (e.g., 0=0) become "true". A "true" is simply discarded, but "false" anywhere causes failure of the whole unification. This is how the "boolean" supertype is used in STAR.

5.2.3. Examples

General function: "f" func (3, 4, "a", "X") (note: "," is right- associative)
List: "a" cons "Y" cons "nil" (note: "cons" is right-associative)
(Numeric) expression: "X" + 4 × "Z" − "Y"
Term: "X" cons "f" func ("a", 4) cons 5+"W" cons "nil"
(note: "func" is infix binary operator)
Bindings: "V" cons "W" eq "a" cons "X"; "Y" eq "W"; "Y" eq "nil"

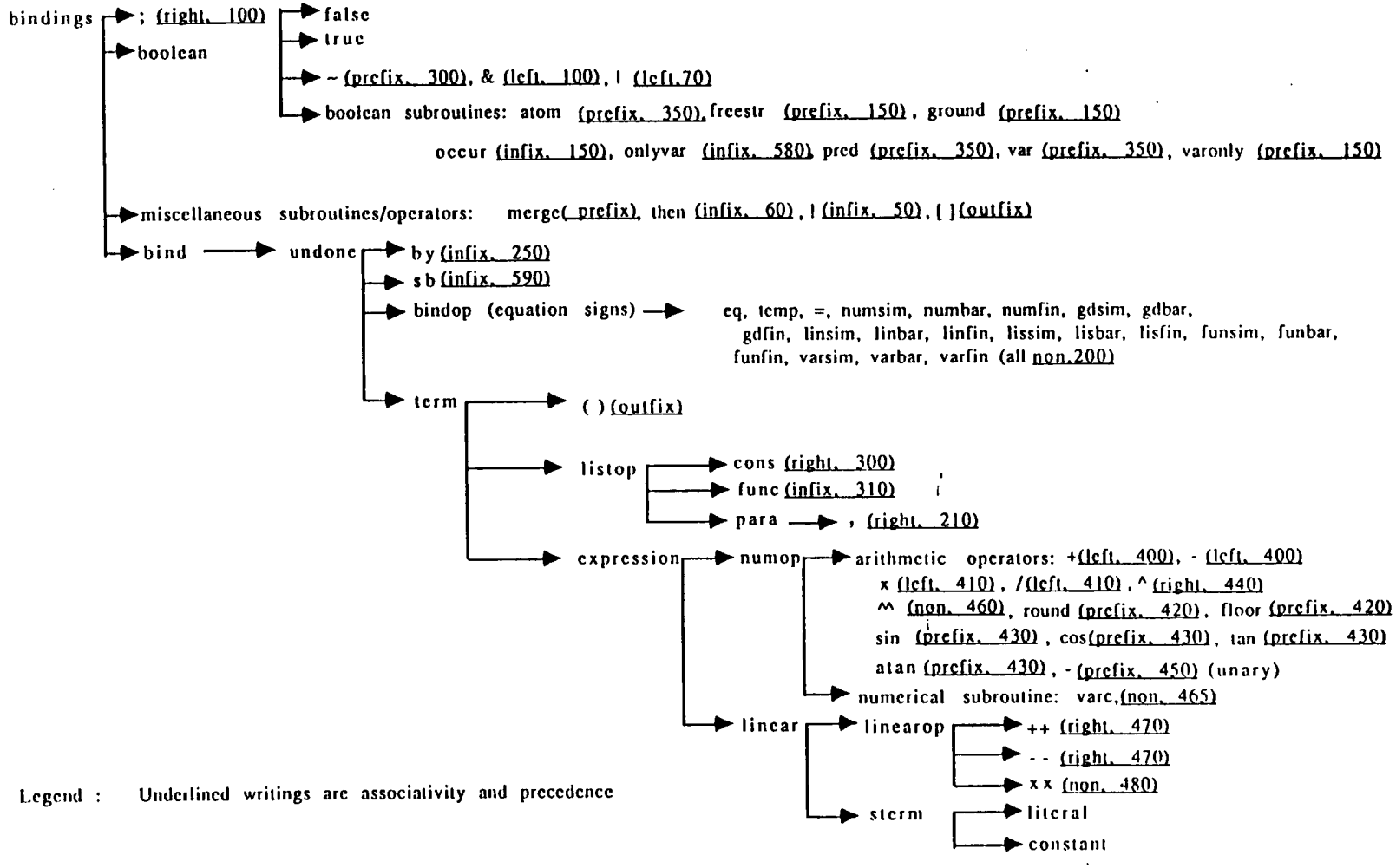

Figure 1. Data type hierarchy for STAR

Legend : Underlined writings are associativity and precedence

5.3. *Stratified Term Rewriting*

Defn. 5.9 — A **typed expression** is a subject expression with the addition that some variables in it may be typed (i.e., followed by a close single quote then the type name).

For example, X'number + Y'variable is a typed expression where X and Y are typed variables.

Defn. 5.10 — Given an expression E and its variable set V, some of which have a type specification, a **substitution** θ for E is the function defined in Chapter 4 except that each typed variable v is matched to a term t of the same type or its subtype(s) (direct or indirect).

Defn. 5.11 — Given a rewrite rule with typed variable(s) in its head H_i , and a subject expression E, we say H_i **matches** a subexpression E_s of E if there is a substitution θ such that $H_i \bullet \theta = E_s$.

Defn. 5.12 — A **stratified term rewriting system** is a quadruple <H, V, S, R> where H is a data type hierarchy, V a set of (typed or untyped) variables, S a set of subject expressions, and R a set of rewrite rules whose heads are typed expressions.

For example,
<H, {x, y}, S, {x'literal + y'literal → x+ +y, x'variable + y'constant → y+x}>
is a stratified term rewriting system where H is described by the diagram in Figure 2, and S is the set of all possible expressions based on H.

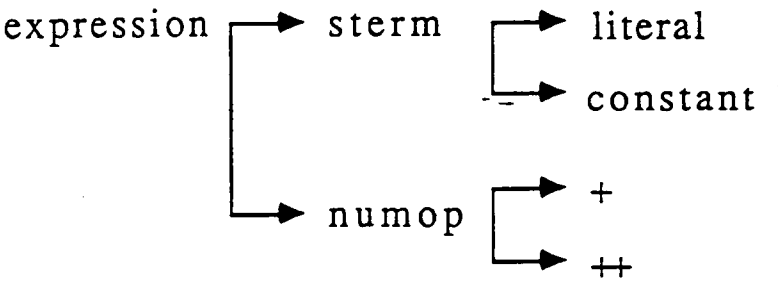

Figure 2. Data type hierarchy for an example

The advantage of a stratified term rewriting over standard term rewriting is that under the same circumstances, a typed subject expression matches fewer rules than its untyped counterpart. For example, the expression "V" + "W" will be rewritten into "V" + + "W" in the example system above, while its standard term rewriting counterpart

<{x,y}, S, {x+y → x+ +y, x+y → y+x}>

would have two different answers, namely "V" + + "W" and "W" + "V".

Remarks

In this thesis we are concerned only with rewrite rules that do not introduce new variables on their right hand sides (i.e., all variables on the right hand side of a rule also occur on the left hand side). Also, hereafter we omit the word "stratified" in the discussion; we refer to non-stratified term rewriting systems when speaking of standard term rewriting systems. Finally, all the right hand sides of rewrite rules have their variables untyped since they are already typed on the left hand sides.

5.4. *Using Stratified Term Rewriting for Arithmetic Unification*

Let us recall the arithmetic unification algorithm from Chapter 4. The goal was to find an arithmetic unifier for the input unifying equations. With stratified term rewriting, however, the first problem is the difficulty of operating on two inconsecutive objects. Or, in other words, it is hard to control the order in which the rewriting takes place. For example, if we have

A; B; C; D; E

where each of A, B, C, D, and E represents an equation, and we want to modify C;D out of the context of the whole list, then we are likely to fail because the system would rewrite A(or E, depending on the associativity of ;) first if there is a proper matching. This serious control problem requires that we modify the general algorithm of Chapter 4.

In order to solve the above problem, we choose to look at two equations at a time to get information necessary for variable bindings. We make sure to check every possible pair of equations (original ones as well as their decendents) — eg,

"3 numsim 5; "X" varsim "Y" "

is a decendent of

""X" cons 3 eq "Y" cons 5".

This way we should be able either to detect an inconsistency in the input or to obtain the arithmetic unifier.

For example, if the input is

"X" eq 4; "Y" eq 5; "X" eq 6

then we would detect an inconsistency between "X" eq 4 and "X" eq 6 by checking every possible pair of equations.

One logical way of doing this kind of checking in the context of stratified term rewriting is to "move" the leftmost equation rightward, checking it against each equation that it "passes by" and do the same for all equations but the last one. There is the obvious problem of indefinitely switching two adjacent equations, however, and the problem of not being able to recognize the last of the equations to be moved. We need some kind of mark to accomplish the task.

5.5. *Classification of Equations*

The idea of using a mark naturally leads to the categorization of equations, an idea that we shall use throughout the remainder of this thesis. We define six **classes of equations.**

Equations in other forms are to be converted to one of these six classes or resolved directly into either true or false.

For example,

$$5 \times \text{"X"} \times \text{"Y"} \text{ eq } 6 \times \text{"W"}$$

would be rewritten into

$$5 \times \text{"X"} \times \text{"Y"} - 6 \times \text{"W"} = 0 \text{ (class } L_6\text{)}.$$

As another example, the equation 3 eq 4 is resolved into "false" directly thus causing the whole list (of equations) to be rewritten into "false".

The following is the list of all six equation classes. The "—" sign simply represents a binding relation (in cases of L_1 to L_5) or an equation (in case of L_6).

- L_1.
  variable — number (eg, "X" = 3)
  variable — atom (eg, "X" = "a")
  variable — ground-list (eg, "X" = "a" cons "nil")
  variable — ground-general-function (eg, "X" = "f" func ("b"))
- L_2. variable — variablized-linear-expression (eg, "X" = 4 × "Y" — "Z")
- L_3. variable — non-ground-list (eg, "Y" = "X" cons "Z")
- L_4. variable — non-ground-function (eg, "X" = "f" func ("W"))
- L_5. variable — another-variable (eg, "X" = "Y")
- L_6. non-linear-expression — 0 (eg, "X" × "Y" + 4 = 0)

We say that $L_i < L_j$, or L_i is of lower class than L_j, for i < j. We also say that an equation is of "class i" if it is in the form of L_i. Roughly speaking, the lower class an equation is in, the closer it is to being solved.

Recall the sets G, L, F, N, and U in Chapter 4. Note the relation of our equation classes to these sets. It is obvious that L_1 corresponds to G, L_2 corresponds to L, L_3, L_4, L_5 correspond to F, L_6 corresponds to N, and the unclassified equations (with "eq" or "temp" as their equation signs) correspond to U.

For each class L_i ($1 \leq i \leq 5$) there are three corresponding equation types. These three types are called "simple equal", "barrier", and "final" respectively. Basically, "barrier" is used after interaction between equations of different classes, and "final" is used after interaction between equations of the same class. A chart for all equation signs can be found in Figure 3.

As an example, look at the list of equations

x eq 4; y eq x cons nil; y eq z.

First we rewrite "x eq 4" into "x numsim 4", "y eq x cons nil" into "y lissim x cons nil", and "y eq z" into "y varsim z". Next we check the first two equations, "x numsim 4" with "y lissim x cons nil", and find that they are consistent, and that x should be substituted by 4 in the second equation. We do the following simultaneously: switch the two equations, substitute x by 4 everywhere in the equation "y lissim x cons nil", and set up the barrier for the equation "x numsim 4". The resulting list of equations after this swapping is:

Equation Type	Description	Simple Equal	Barrier	Final
L_1	variable - number	numsim	numbar	numfin
	variable - atom	gdsim	gdbar	gdfin
	variable - ground list	gdsim	gdbar	gdfin
	variable - ground fcn.	gdsim	gdbar	gdfin
L_2	variable - linear	linsim	linbar	linfin
L_3	variable - list	lissim	lisbar	lisfin
L_4	variable - function	funsim	funbar	funfin
L_5	variable - variable	varsim	varbar	varfin
L_6	non-linear = 0	=	=	=
Input	anything - anything	eq		
Interme-diate	argument list - argument list	temp		

Figure 3. Equation signs for classified equations

y lissim 4 cons nil; x numbar 4; y varsim z.

The barrier is necessary here because otherwise the system would have rules to match "y lissim 4 cons nil; x numsim 4" and get into an infinite loop.

For the same reason, we have type "final" equation signs for equations of the same type. Suppose we have "y eq a" in place of "y eq x cons nil" in the above example, then the resulting list would be:

y gdsim a;x numfin 4; y varsim z

after the first swapping.

Now it should be clear that the three types of equation signs for the same class of equations are merely a finer categorization of equations. When we write rewrite rules, we often have to distinguish between different types of equations of the same class.

There are intermediate forms of equations that can be easily solved directly or transformed into one of the L_i's. Sometimes we purposefully want an equation in an untyped form so that we can "clean it up" before it is involved in rule matching.

For example, when we have a non-linear equation 4*X-X*X=0 interacting with an equation of the form X=4, we know (after checking consistency) that X is the only variable in the non-linear equation and that the equation would become linear (in fact, either "true" or "false"). Therefore we rewrite the substituted equation in the form "4*4-4*4 numsim 0", for example, so that it will not be treated as class L_6 and as a consequence will be rewritten to either "true" or "false" before it interacts with other equations (like it would if "numsim" were to be "="). This way we save some (maybe many) rewrite steps.

Along with the classification of equations by means of distinct equation signs, we have also set up priority rules for checking consistency between any two equations so that swapping happens in only one direction but not both. For example, if the original list in the above example were

y eq x cons nil; x eq 4; y eq z

then we switch the first two equations without changing their content. This way the lower class equations can move to the leftmost position before they are checked. This is necessary because otherwise they will not be checked with higher class equations that were on their left.

The general rules for checking consistency and swapping are as follows:
if the equation on the left has higher class than the one on the right, then switch their positions without doing anything else;
if the equation on the right has higher class than the one on the left, then switch their positions, make substitutions where applicable, and use the barrier equation sign for the lower class equation after switching;
if the two equations under check are of the same class, then do the proper substitution(s) while switching their positions and use the final equation sign for the second equation (after switching).

Due to substitutions, there are cases where new (classes of) equations evolve after one check. In this case, we have no general rule and have to consider specific situations.

Due to the lack of a global name space and of a method to substitute the variables automatically by the values they are bound to everywhere in the list of equations, we need

a subroutine to perform variable substitution. We introduced the operators "by" and "sb" to do this.

For example, when we have

x numsim a; y lissim z

where x and y are variables and z is a non-ground list that contains x, we must substitute all occurrences of x in z by a (which is a number by definition). We rewrite the whole equation string into

y lissim (x by a) sb [z]; x numbar a.

We can see the syntax of "by" and "sb" very clearly here. The term "(x by y) sb [z]" simply means to substitute all occurrences of x in z by y, where x is a variable and y and z are any terms.

5.6. STAR's Algorithm

The data structure part of the algorithm has already been presented in Section 5.2. Here we only give the rule part of the algorithm. Of course, the algorithm is far more abstract than the actual program, STAR; that is, we shall give only the outline of the algorithm here.

The algorithm consists of several modules as shown in Figure 4. We will present each module along with its explanation.

It is helpful (if not necessary) to know how STAR matches an object in the input (or subject expression). The basic idea is to search, from left to right, for the longest pattern possible and rewrites it. For the specific kinds of input we may have, the longest possible pattern would be two consecutive equations. The next longest pattern would be one equation. The next one would be the left hand side of an equation, the next would be a part of the left hand side of an equation, so on and so forth. Of course, the matching only happens to the leftmost possible pattern.

An example will help understand the idea. Suppose we have the following list of equations:

X+4 cons nil eq Y; Z eq 5; W eq 6

Searching from left to right, we realise that the first equation is not completely "solved", i.e., could be further rewritten. When we look at the second equation, however, no rule will match the subexpression, "X+4 cons nil eq Y; Z eq 5". Therefore we look for a rule that will match the first equation alone. Suppose we fail again, then the next object would be "X+4 cons nil", since "eq" is the outermost operator in that equation. Suppose this attempt fails again, then the next target would be "X+4" for "cons" is the outermost operator for the subexpression "X+4 cons nil".

I. Conversion of the Intermediate Forms

This part of the system rewrites each input (or intermediate) equation of the form "LT eq RT" into its internal representation, i.e., into one of the forms L_i ($1 \leq i \leq 6$), or into some immediately solvable form and solve it (i.e., rewrites it into "true" or "false").

It also converts equations of the form "term1 temp term2" where "term1" and "term2" are augument lists for some general functions. This kind of equation is broken down on a one-to-one basis. More specifically,

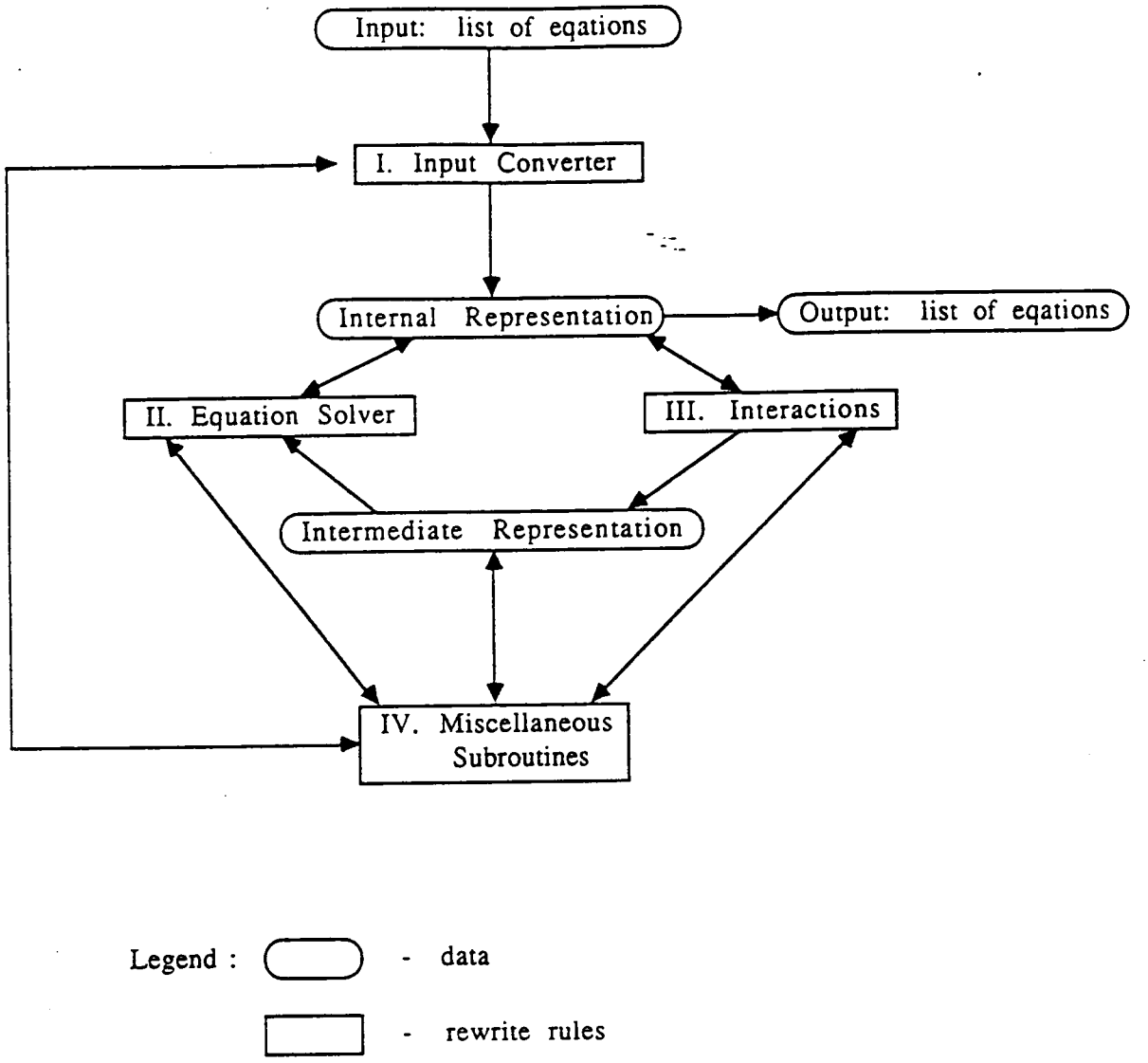

Figure 4. Flow chart for STAR

(arg1, rest1) temp (arg2, rest2)

is rewritten into

arg1 eq arg2; rest1 temp rest2.

The following is an account of how to convert equations of the form "LT eq RT".

1. If both LT and RT are functions in the form a(b) and c(d) respectively, where a = c, a is either "cons" (list function name) or any other general function name, and b and d are argument lists, then set up a new equation (or equations) between b and d, in place of "LT eq RT"; if a ≠ c, then fail.
2. If one of LT and RT is a general function or list, and the other is a non-variable (e.g., a constant), then fail; if the other is indeed a variable that does not occur in the function, say LT, then bind RT onto LT through an appropriate equation; if LT occurs in RT then fail.
3. If either LT or RT is a numeric expression, and they are not both variables, then rewrite "LT eq RT" to "LT - RT = 0".
4. If either LT or RT is a variable, say LT is, and RT is a non-numeric ground term, then replace "LT eq RT" by "LT gdsim RT".

II. Linear Equation Solver

This part of the system solves or simplifies numeric equations. Basically it performs the same function as the reduction algorithm in Chapter 4 except that we have to create linear terms from "single occurrence" of variables (by intuition). Therefore we only present how to create linear terms here.

- When a variable is connected by "×" to a number, rewrite this variable by its linear form (e.g., 1 ×× X + + 0 is the linear form of X).
- When a variable is connected by "+" to anything, rewrite it by its linear form.
- When two variables are connected by "+", rewrite both of them by their linear forms respectively.

This way it is guaranteed that only "single occurrences" of variables become linearized. For example, in the context of 4 × X × 5 × Y - 6 × Z + 2 × W, although the first term is not linear, we would still get a linearized X (since × is right-associative, X is the one that is connected with the constant 20). But Y would not be linearized (since it is connected with linearized X), so the first term remains non-linear in the form. The second and the third terms, however, are linearized and become -6 ×× Z + + 0 and 2 ×× W + + 0, respectively. The final result is:
2 ×× W + + -6 ×× Z + + 0 + 20 × 1 ×× X + + 0 × Y.
Note that the precedence ordering among the arithmetic operators above is:

(unary) - > ×× > + + > × > +.

Therefore the implied ordering for the final expression above is:
((2××W)+ +((-6××Z)+ +0))+(20×((1×× X)+ +0))×Y.

III. Interactions Between Consecutive Equations

Here we describe how consecutive equations can be swapped. These rules try to match two consecutive equations in the subject list. We use E_i to denote an equation of class L_i, $1 \leq i \leq 6$.

1. If there are equations of the form

$$E_i;\ E_j$$

where i > j, then simply rewrite them into

$$E_j;\ E_i.$$

2. If there is a pattern

$$E_i;\ E_6$$

where $1 \leq i \leq 5$, then basically we check the consistency of variables occurring in these equations, perform the appropriate substitutions, and swap the equations.

Detailed Explanations Suppose E_i is of the form

var equal term

where "term" may be a number, a ground atom, a list, a function, a variable, or a linear expression and "equal" a classified equation sign corresponding to "term".

Recall that E_6 should be a non-linear equation in the form q = 0. Check whether "var" occurs in q. If it does, and "term" is (or can be) a numeric value, then rewrite the original pattern into

q • {var = thing} = 0; var equal' thing

where "equal'" is of type "barrier" if "equal" was of type "simple equal" or "barrier", and "equal'" is of type "final" if "equal" was of type "final". If "var" occurs in q and "term" can not take a numeric value, then the rewrite result is "false", which eventually makes the whole list (of equations) "false". If "var" does not occur in q then the result is

E_6; *varequal* term

where "equal'" is as described above.

If the pattern is

$$E_6;\ E_6$$

then do nothing. This is because we cannot get any new information in terms of equations from two non-linear equations, which is determined by the fact that we cannot "recognize" the relations of two non-linear terms.

Example

For example, if we have

$$3 \times X + X \times Y = 0;\ 2 \times X + X \times Y = 0$$

we cannot "cancel" the term X × Y by any means in STAR, so there is no way we can get $5 \times X = 0$ thus X = 0. As we can easily see, we can recognize the pattern of the non-linear terms ourselves.

3. If there is a pattern

$$E_i;\ E_5$$

where $i \leq 5$, then basically we check the relationship between the variables involved, and either simply swap the equations or perform proper substitutions or generate a new equation accordingly.

Detailed Explanations

Suppose E_5 and E_i are in the form

var1 equal1 var2
and
var3 equal2 term

respectively, where "equal1" is "varsim", "varbar", or "varfin"; "term" may be a number, an atom, a linear expression, a list, or a function, and "equal2" is a corresponding equation sign. Recall that "var2" must be alphabetically greater than "var1".

If the two equations are identical, then discard one of them and keep the other.

If "var3" is "var1" or "var2", then rewrite

$$E_i; E_5$$

into

var equal term; var3 equal1′ term

where "var" is either "var1" or "var2" and "var3" is the other; "equal1′" is either the "barrier" or the "final" type for "equal1"; "equal" is the "simple equal" type for the same class as "equal1". There is an exception here when i = 2, i.e., when the first equation was in the form

var1 equal1 linear1,

and "var3" is "var1". In this case, the first resulting equation, instead of "var equal term", would be

term — var2 = 0.

If "var3" is different from both "var1" and "var2", then simply swap the two equations and change "equal1" to "barrier" or "final" if it was not before.

Examples

For example, if the two equations are:

X numsim 4; Y varbar X,

then we have

Y numsim 4; X numfin 4.

As another example,

X linsim Y — 3; X varsim Z

would result in

Y — 3 — Z = 0; X linbar Y — 3.

4. If there is a pattern

$$E_i; E_4$$

where $1 \leq i \leq 4$, then basically we check consistency between variables in these equations, perform substitutions where applicable, and swap the equations. New equations may be generated in this case.

Detailed Explanations

Suppose E_i and E_4 are in the form

var1 equal1 term; var2 equal2 function

where "equal1" and "term" correspond to an equation of class L_{i_1} "equal2" is

"funsim", "funbar", or "funfin", and "function" is in the form "functor func (arglist2)".

- If "var1" is not "var2", and neither "var1" nor "var2" occur in the other, then simply swap the two equations and change "equal1" to "barrier" or "final" if it was not before.
- If "var1" is "var2", and "term" is not a function or does not have the same functor as "function", then fail; if "term" is a function in the form "functor func (arglist1)", then the resulting equations are:

  arglist1 temp arglist2; var1 equal1' term

  where "equal1'" is the "barrier" form for "equal1" (unless "equal1" was of "final" form in which case "equal1'" is, too), and "temp" is one of the unclassified equation signs.
- If "var1" is not "var2", but either "var1" occurs in "arglist2" or "var2" occurs in "term", (but not both - in which case fail;) then the result is:

  var2 equal2' function'; var1 equal1' term'

  where "equal2'" is the "barrier" or "final" form of "equal2" (in case no substitution occurred to the second equation), or "simple equal" type sign for a lower class equation (in case "var1" is the only variable in "arglist2" and "term" is a constant term); "term'" and "function'" are the substituted form for "term" and "function", respectively. "equal1'" is the "barrier" or "final" type for the same class of equation as "equal1".

Example

For example, the equation sequence

V lissim X cons nil; X funsim f func (Y)

yields the new sequence

X funsim f func (Y); V lisbar f func (Y) cons nil.

As another example, the equations

V numsim 4; X funfin f func (V)

yields the new equations

X gdsim f func (4); V numfin 4.

As yet another example, the sequence

V funsim f func (X); X funbar f func (V)

fails (yields "false") because "V" and "X" occur in each other.

5. The pattern is $E_i; E_3$ where $1 \leq i \leq 3$, then basically we check the consistency between all variables involved, perform substitutions where applicable, and swap the two equations. New equations may be generated in this case.

Detailed Explanations

Suppose that E_i is of the form

var1 equal1 term1,

and E_3 is of the form

var2 equal2 list2.

- If "var1" and "var2" are different, and none of them occur in the right hand side of the other equation, then simply swap the two equations and change "equal1" to "barrier" or "final" if it was not before.
- If "var1" and "var2" are different, and they both occur in the right hand side of the other equation, then fail (infinite self referencial).

- If "var1" and "var2" are different, and one of them occurs in the other equation, then the following new sequence of equations are generated:

  var2 equal2' list2'; var1 equal1' term1'

  where "list2'" and "term1'" are the possibly substituted "list2" and "term1" respectively; "equal1'" is the "barrier" or "final" form for "equal1", and "equal2'" is the "barrier" or "final" form for "equal2" (when the possible substitution did not make "list2" a constant), or "simple equal" form (when the substitution makes "list2" a constant).
- If "var1" is the same as "var2", and "term1" is not a list, then fail
- If "var1" and "var2" are the same, and "term1" is in the form "h1 cons t1" while "list2" is in the form "h2 cons t2", then the new equations

  h1 eq h2; t1 temp t2; var1 equal1' term1

  are generated, where "equal1'" is the same as in the case above. Note that the first two equations in the new sequence will have to be converted before further interactions take place.

Examples

a. The sequence

V numfin 5; X lisfin V cons nil

yields

X gdsim 5 cons nil; V numfin 5.

b. The sequence

V lissim X cons nil; X lisbar V cons nil

fails.

c. The sequence

W lisbar X cons a cons nil; W lissim Y cons a cons nil

yields

X eq Y; (a cons nil) temp (a cons nil); W lisbar X cons a cons nil.

6. The pattern is E_i; E_2 where $1 \leq i \leq 2$, then basically we check occurrence relations of variables and perform the appropriate substitutions where applicable, and either swap the equations or generate new equations according to the case.

Detailed Explanations

Suppose that E_i is of the form

var1 equal1 term1,

and E_2 is of the form

var2 equal2 linear2,

and "equal2" is "linsim", "linbar", or "linfin".

- If "var1" and "var2" are different and neither of them occurs in the other equation, then simply swap the two equations and change "equal1" to "barrier" or "final" form if it was not before.
- If "term1" is not a numeric value (i.e., a constant literal, a ground list, or a ground function), and either "var1" is the same as "var2" or "var1" occurs in "linear2", then fail.
- If "var1" and "var2" are the same, then the new sequence is:

  linear2 - term1 = 0; var1 equal1' term1

  where "equal1'" is the proper alternative for "equal1" as described before. Since both "linear2" and "term1" are linear expressions (with "term1" possibly being a number), the first equation in the new sequence would eventually be resolved

into an equation of the form "var equal term" where "var" is the alphabetically smallest variable in "linear2" and "term1", and "term" is either a constant number or a linear expression.

- If "var1" is different from "var2", and one of them occurs in the other equation, then the new sequence will be:

  var2 equal2′ linear2′; var1 equal1′ term1

  (if "var1" occurs in "linear2"), or

  var2 equal2 linear2; var1 equal1″ term1′

  (if "var2" occurs in "term1"). "var2 equal2′ linear2′" is the result of substitution of "var1" in "linear2", and may be of class L_1 or L_2 depending on the case. "var1 equal1″ term1′" is the result of substitution of "var2" in "term1" and may be of class L_1 or L_2. "var1 equal1′ term1" is basically the same as "var1 equal1 term1" except that "equal1′" has "barrier" or "final" type while "equal1" could be any of the three type equation signs.

Examples

The sequence

X gdsim a; W linfin 1 ×× X + + 4

fails.

The sequence

X numsim 4; X linbar 2 ×× Y + + 8

yields

2 ×× Y + + 8 - 4 = 0; X numfin 4

which eventually would yield "Y numsim -2" from the first equation.

7. The pattern is E_1; E_1, which can be more specifically written in the form:

   var1 equal1 term1; var2 equal2 term2.

   "term1" and "term2" here are both constants (numeric or non-numeric). Since obviously no substitution is possible in this case, we only check if there is any conflict between the two equations in case "var1" is the same as "var2", and either swap the equations (changing "equal1" into its "final" form) or fail.

   Example

   If the sequence is

   X numbar 4; Y gdbar b

   then the result is

   Y gdbar b; X numfin 4.

   If the sequence is

   X gdfin b; X gdsim a,

   then fail (yields "false").

IV. Miscellaneous Utilities

During the course of rewriting, there are many "minor" things that need to be checked. The following is a complete list of all such utilities. They are mostly self-explanatory; but in case they are not, we shall give a comprehensive explanation.

1. **atom:**
   call format: atom(literal)

2. **freestr:**

call format: freestr(term)
The purpose of this check is to make sure no non-numeric terms are involved in numeric manipulations. For example, freestr(a+4) would give "false".

3. **ground**
call format: ground(term)

4. **occur**
call format: var occur term

5. **onlyvar**
call format: var onlyvar term
The purpose of this check is to see if "var" is the only variable in "term". This is useful upon substitution — as we saw in the algorithm.

6. **pred**
call format: pred(literal)
This checks to see whether "literal" is a pre-defined general function name.

7. **sb/by**
call format: (var by term) sb [subject-term]
This is our substitution routine. It substitutes all occurrences of "var" in "subject-term" by "term".

8. **var**
call format: var(literal)
This checks whether "literal" is a pre-defined variable name.

9. **varc**
call format: var1 varc var2
This compares "var1" and "var2" lexically, and returns 0 (if "var1" is smaller than "var2"), 1 (if "var1" is the same as "var2"), or 2 (if "var1" is greater than "var2").

10. **varonly**
call format: varonly[term]
This checks to see if all literals in "term" are variables. This is called by "freestring".

5.7. Summary

In this chapter we formally introduced stratified term rewriting and presented a stratified term rewriting system, STAR, for implementing arithmetic unification.

In STAR, the commutativity law is achieved by solving equations. When solving equations, we use a systematic procedure on each equation to obtain a "minimal" or reduced equation and then find the answer if it is available. An example will help illustrate the idea.

If we want to unify X+3 with 3+X, STAR creates an equation (X+3) - (3+X) = 0 instead of applying the commutativity law. Then the reduction algorithm systematically derives the equation "0=0" and therefore "true" in a finite number of steps.

If non-linear terms are involved, however, commutativity may not be fully implemented. For example, if we had "X^2" or "X &mul. Y" in place of "X" in the example above, the reduction algorithm would not be able to derive the same answer, and STAR would return an unsolved non-linear equation.

Actually, we do not pursue reduction of non-linear terms because the cost is very high (as explained in Chapter 3). If we could find more efficient implementations (as suggested in Chapter 8), however, we could add non-linear reduction to our system. Terms like "$X^2 - X^2$" and "X &mul. Y - Y &mul. X" could then be reduced.

The equation signs introduced in this chapter solve a different set of problems. Given the current implementation of Bertrand, our classification of equation signs implements binding values to variables in the unification process and guarantees the termination and consistency of unification. In the next chapter we will prove that STAR terminates and is confluent, among other properties.

6. Confluence and Termination

In this chapter we discuss properties of STAR from a more theoretical point of view. There are 3 sections in this chapter. Section 1 defines our terminology. Section 2 discusses termination-related properties. Section 3 discusses confluence-related properties.

6.1. Terminology

Defn. 6.1 — For a subject expression E and a set of rewrite rules R, if E can be transformed (or rewritten) into E_1, which then is transformed into E_2(not necessarily by a different rule), and so on, then we have a **reduction sequence** $E \rightarrow E_0 \rightarrow E_1 \rightarrow E_2 \rightarrow \ldots$, and E_i is **reduced (or rewritten) to** E_{i+1} , $i \geq 0$. If for some $i \geq 0$, E_i cannot be further reduced, E_i is called **irreducible**.

Defn. 6.2 — If for every rule in a term rewriting system S, the similarity relation between the head and body is equality, then S is said to be **sound.**

Defn. 6.3 — The reduction sequence $E_0 \rightarrow E_1 \rightarrow \ldots \rightarrow E_i \rightarrow \ldots$ **terminates** if there exists a subject expression E_n in the sequence that is irreducible. E_n is called the **normal form** of E_0. A term rewriting system is **terminating** if all of its reduction sequences terminate.

Defn. 6.4 — For a given set of rewrite rules R, take any subject expression E. If E has more than one reduction sequence all of which terminate and give the same normal form E_n, or if E has only one reduction sequence and it terminates with a normal form E_n , or if E is irreducible (in which case E is its own normal form), then R is **confluent (or deterministic).**

Defn. 6.5 — A term rewriting system is **overlapping** if the head of some rule matches a non-variable subterm in the head of another (possibly the same) rule. The system is **non-overlapping** otherwise.

Defn. 6.6 — A term rewriting system is **left-linear** if no left hand side has more than one occurrence of the same variable.

Defn. 6.7 — A term rewriting system is **locally confluent** if any subject expression can be matched by at most one rewrite rule.

A locally confluent term rewriting system is **confluent** if it is terminating [Bund ??]

6.2. *Termination*

In this section we give the proof of termination for STAR.

First we give some definitions that are used for this proof.

Defn. 6.9 — An **equation list** and its **length** are defined inductively as follows.

1. "true" is an equation list of length 1.
2. "false" is an equation list of length 1.
3. an equation of the form

   term1 equal term2

   is an equation list of length 1, where "equal" can be any one of the equation signs listed in Figure 3 in Chapter 5, and "term1" and "term2" can be any valid subject expression defined in Chapter 4.
4. "equation-list; equation" is an equation list of length n+1, where "equation-list" is an equation list of length n, and "equation" is either one of the basic equation lists defined in 1, 2, and 3 above.

Defn. 6.10 — In an equation list that is being rewritten, the first equation is called the **head** of the list, and the last equation the **rear**. If the sublist containing the first n ($\geq$ 0) equations of the list is irreducible but the one containing the first n+1 is reducible, then we call this sublist of length n the **minimal head** of the list.

Defn. 6.11 — An equation with the equation sign "eq" or "temp" is called **newly-founded**.

Defn. 6.12 — An equation of one of the six classes defined in Chapter 5 is called **classified**.

Defn. 6.13 — If a list of classified equations, after finite rewrites, results in newly-founded equations along with (possibly no) non-newly-founded ones, then the list is called **productive**. A list of classified equations that does not result in any newly-founded equations in a finite number of steps is called **simple** or **non-productive**.

Defn. 6.14 — Given an equation E in the form

variable equal expression

where "variable" is a variable, "equal" an equation sign, and "expression" an expression that has substitutions embedded in it, the substitutions embedded in "expression" are called **local substitutions**. Informally, local substitutions are merely simplifications, or reductions, for the right hand side of an equation.

For example, the equation

X numbar (Y+4) sb [Y by 5]

has local substitutions in it. Note that although the subject expression ("y+4" in this case) has a variable in it, the equation sign is for ground binding. This is because the binding will be ground after all substitutions are carried out - the rewrite rules are designed to discover situations where the only variable involved is to be substituted even before the actual substitution action takes place.

6.2.1. Properties and Theorems

Now that we have defined some concepts applicable to term rewriting systems, we are ready to see some properties and theorems of STAR.

Property 1. In an irreducible equation list that is not "true" or "false", every equation must be an irreducible classified equation.

Proof.
For any equation list E_1; ...; E_n, if some E_i ($1 \leq i \leq n$) is classified and reducible, then some rewrite rule would apply to E_i therefore further rewriting the list. If some E_j ($1 \leq j \leq n$) is not classified, then it must be in the form "true", "false", or newly-founded. Further rewriting is possible in all three cases provided that $n > 1$ (in the first two cases). Therefore, each E_i ($1 \leq i \leq n$) in the list must be an irreducible classified equation.

Property 2. In an irreducible equation list, every equation must have an equal or lower class than that of the equation to its left. The rear has the lowest class.

Proof.
With Property 1, we may consider only the classified equation lists. Take any classified equation list E_1; ...; E_n, if some consecutive equations "E_i; E_{i+1}" ($1 \leq i \leq n-1$) are such that the class of E_i is lower than that of E_{i+1} , then some rewrite rule would match "E_i; E_{i+1}" - since the rewrite rules are so designed. Therefore, if an equation list is irreducible, then every equation in it must have an equal or lower class than that of the one to its left. Consequently the rear has the lowest class.

Property 3. In an irreducible classified equation list, each equation of class L_1 to L_5 has a distinct left hand side variable.

Proof.
It is sufficient to show that the interactions between equations are "complete" in the sense that each equation in the final list has interacted with all the rest.

Lower class equations get shifted to the left (lowest to the leftmost) of the list and move rightward, interacting with equations that have the same or higher class. In case new, low class equations result, they move leftward then rightward just like the original equations. The final list, as shown by Property 2, occurs in a descending order by equation class.

This completeness is guaranteed by the 625 rewrite rules that cover all possible interactions between two equations. (There are 12, 3, 3, 3, 3, 1 kinds of equation signs for class L_1, L_2, L_3, L_4, L_5, L_6 equations, respectively. So there are 25 cases for each of the two interacting equations.)

Property 4. In an irreducible equation list, no left hand side variable (of L_1 to L_5 equations) occurs in any right hand side (of L_1 to L_5 equations) or (left hand side of) L_6 equations.

Proof.
This is guaranteed by the completeness of the rewrite rules, and all possible substitutions must take place before rewriting has quiesced.

Property 5. For a list of equations

$$List1 = E_{11}; ...; E_{1n},$$

suppose that the lowest class equation of the classified equations in List1 is C1. For any equation list List2 derived from List1 (in a finite number of steps), suppose that the lowest class equation of the classified equations in List2 is C2. Then C2 must be no greater than C1, unless "false" occurs.

Proof.
Consider an arbitrary equation list List1 with its corresponding C1. Consider all values that C1 may take.

Case 1. C1 is of class L_1. Since there are no right hand side variables in equations of class L_1, no substitutions can be applied to such equations. Therefore, L_1 equations remain unchanged during rewriting - unless "false" occurs. In other words, unless "false" occurs, for any List2 derived from List1 in a finite number of steps, C2 will be L_1. Therefore, C2 is equal to C1.

Case 2. C1 is of class L_2. Suppose E_{1i} $(1 \leq i \leq n)$ is of class L_2 which is in the form "var1 equal linear-expression". During rewriting, variables on the right hand side of E_{1i} may be substituted by linear expressions that may or may not be ground. When all variables have been substituted by ground expressions, E_{1i} becomes a class L_1 equation; otherwise E_{1i} remains an L_2 equation - unless "false" occurs, which is caused by non-numeric substitutions or variable binding conflicts. Therefore, after one rewriting step, E_{1i} will be of class L_1 or L_2. This is also the case after any finite number of rewriting steps. In other words, for any List2 derived from List1 in a finite number of steps, its C2 will be L_1 or L_2 unless "false" occurs. Therefore C2 is of class no greater than C1.

Case 3. C1 is of class L_3. With arguments similar to that of case 2, we can show that L_3 equations in List1 become L_1 or L_3 after any finite number of steps of rewriting, unless "false" occurs. That is, for any List2 derived from List1, its C2 is L_3 or lower unless "false" occurs.

Case 4. C1 is of class L_4. As with case 3, the property holds.

Case 5. C1 is of class L_5. (Recall that L_5 equations are in the form "var1 equal var2".) We have only two possible interactions to consider. L_5 with L_5 and L_5 with L_6. In both cases the L_5 equation remains unchanged, unless "false" occurs or unless the right hand side variable is substituted by an expression, in which case the L_5 equation becomes L_1, L_2, L_3, $\vee L_4$, depending on the case. Therefore, after one rewrite step, the lowest clases would be L_5 or lower unless "false" occurs. This holds for any number of finite rewriting steps, too.

Case 6. C1 is of class L_6. No rewriting takes place (according to the rewrite rules), therefore the property holds.

For all 6 cases above, we have proved Property 5.

Now we present some theorems related to STAR.

Theorem 6.1 The rewriting of a simple equation list terminates in a finite number of steps.

Proof.
Use induction on the number of equations in the list.

Base case. There is one equation in the list.

If the equation is of class L_6 , then it may be rewritten to "false," "true," a linear variable binding (class L_2 equation), or a simplified or irreducible non-linear numerical equation (class L_6). Rewriting terminates in all these cases.

If the equation is one of the classes L_1 through L_5 , then according to STAR's algorithm, it must have been derived from a newly-founded equation or an interaction of two classified equations. In both cases, the rewrite rules are designed such that the equation remains the same except for the local substitutions that should terminate in a finite number of steps.

Therefore, rewriting always terminates if there is only one equation in the list.

Induction hypothesis: Assume that rewriting terminates for a non-productive list of n equations in a finite number of steps. We will prove that rewriting terminates in a finite number of steps for a non-productive list of n + 1 equations as well.

Note that if a list is non-productive, then all sublists and lists in intermediate steps must be non-productive, too. All interactions applied must also rewrite two equations into two (as opposed to three or any other number) new classified equations, and some "true" or "false" equation may result during rewriting. Therefore, the number of equations in the list will either decrease or remain the same; or, in the extreme cases, "true" or "false" results which makes the equation list "trivial."

Suppose we have a non-productive list of n + 1 equations as follows: E_1; ...; E_n; E_{n+1} . The sublist E_1; ...; E_n is non-productive and therefore would either result in an irreducible equation list E_1^1; ...; E_n^1 or in E_1^1; ...; E_m^1 where m < n. In the second case, the new list E_1^1; ...; E_m^1; E_{n+1} is non-productive and has n or fewer equations in it. Therefore, by the induction hypothesis, rewriting terminates in a finite number of steps. Let us further investigate the first case in greater detail.

The next rewrite applies to " E_n^1; E_{n+1} ", and the result would be E_{n+1} ' ; $E_n^{1\prime}$. Note that the class of E_n^1 is the lowest of the equations E_i^1 , $1 \le i \le n$, in the list. The next rewrite transforms the non-productive list E_1^1; ...; E_{n+1}' into E_1^2; ...; E_m^2 according to the algorithm. If m < n, then the new list has no more than n equations, and therefore rewriting terminates with a result satisfying the 5 properties. Otherwise (m = n), rewriting continues, and the next possible step would rewrite the equation pair E_n^2; $E_n^{1\prime}$.

Case i. If E_n^2 has an equal or higher class than $E_n^{1\prime}$, then E_{n+1}' must have had a higher class than E_n^1 (according to properties 2 and 5). Our algorithm guarantees that no rule applies to E_n^2; $E_n^{1\prime}$ - intuitively this is because $E_n^{1\prime}$ has already fully interacted with E_{n+1}' and E_i^1 ($1 \le i < n$), and E_n^2 does not have any new information for $E_n^{1\prime}$. Rewriting terminates at this point and all five properties hold for this case.

Case ii. If E_n^2 has a lower class than $E_n^{1\prime}$, however, the equation pair E_n^2; $E_n^{1\prime}$ does interact, and the entire list is rewritten to E_1^2; ...; E_{n-1}^2; $E_n^{1\prime\prime}$; $E_n^{2\prime}$ in one step.

Now, further rewriting takes place and transforms E_1^2; ...; $E_n^{1\prime\prime}$ into E_1^3; ...; E_m^3 where m ≤ n. We are in a similar situation as before except that the rear ($E_n^{2\prime}$) now has a lower class than the previous rear ($E_n^{1\prime}$). This repetition continues as long as the rear equation does not have the lowest class. There is a lower bound to the classes (L_1), however, and so this repetition must stop after a finite number of iterations. The resulting list satisfies all five properties.

We have just proven that rewriting terminates for a non-productive list of n+1 equations. Therefore Theorem 6.1 is true for all non-productive lists with a finite number of equations.

Theorem 6.2 If a simple (i.e., non-productive) list of equations becomes productive after a classified equation is attached to the end, then the new list can be rewritten into a simple list in a finite number of steps.

Proof.
Use induction on the length of the simple list considered.

In this proof let us use the following representation for classified equations where applicable. Instead of giving names to the equations involved, i.e., E_i, we refer to equations by their left hand side variables (i.e., var equal expression where "equal" is the equation sign and "expression" is the right hand side). In addition, since we will not be concerned with specific equation classes/types, we will use the uniform "=" in place of all equation signs (in classified equations) unless stated otherwise. Therefore a classified equation will be rewritten as: $x_4^1 = ...$ (instead of E_4^1). We also use the notation {EQ;|TEMP;} to represent sublists of newly- founded equations.

Base case: to a simple list of length 1 (i.e., a single classified equation) is attached another classified equation that makes the entire list (of 2 equations) productive. We will show that rewriting terminates in a finite number of steps.

The only way this list can be productive is the case where both equations have the same left hand side variable, and their right hand sides are of the same type which is either a list or a general function (be they ground or not). (This is because that is the only way a newly-founded equation can be generated during interactions between classified equations. Recall that a productive list is one that contains only classified equations and that generates newly-founded equations during rewriting.)

After one rewrite step, the list becomes:

{EQ;|TEMP;} var = ...

The following rewrites will take place on the first equation of the list. Such rewrites will either terminate with a "true" or "false" or generate a classified equation as the first equation with possibly no new newly-founded equations. The sublist {EQ;|TEMP;} therefore becomes headed by a classified equation in a finite number of steps. Further explanation of this point can be found in relevant parts of Theorem 6.5's proof below.

We have just shown that a newly-founded equation can be absorbed by the head in a finite number of steps. Here by "absorb" we refer informally to the phenomenon that a non-classified equation (newly-founded in this case) attached to a classified equation list eventually becomes part of the classified equation list.

Since newly-founded equations are generated from list or function structures (when comparing corresponding arguments), and the list or function structures are all finite, we can have only a finite number of such newly-founded equations for each sublist {EQ;|TEMP;}. Therefore they can be absorbed one by one in a finite number of steps, and thus the original list of length 2 will be rewritten into an irreducible list in a finite number of steps.

Therefore, rewriting terminates in a finite number of steps for the base case.

Induction hypothesis: a simple list of length n-1 attached by an equation that makes the entire list productive can be rewritten into an irreducible list in a finite number of steps. We will prove that the same is true with a simple list of length n.

Let $E_1; ...; E_n$ be a simple list and E_{n+1} a classified equation such that $E_1; ...; E_n; E_{n+1}$ is a productive list. From Theorem 6.1 we know that $E_1; ...; E_n$ can be rewritten into an irreducible list $E_1^1; ...; E_m^1$ in a finite number of steps, where $m \leq n$.

Since $E_1; ...; E_n; E_{n+1}$ is productive, E_{n+1} must be of class L_1 (ground list or ground function), L_3 , or L_4 , and its left hand side variable must be the same as that of some E_k^1 where E_k^1 has the same equation type (i.e., either list or function) as that of E_{n+1} , and E_k^1 is the only such equation among E_i^1 $(1 \leq i \leq n)$ (otherwise the list $E_1; ...; E_n; E_{n+1}$ would not have been productive).

The next rewrite step transforms $E_m^1; E_{n+1}$ into $E_{n+1}'; E_m^{1\prime}$. Rewriting applied to the list $E_1^1; ...; E_{n+1}'; E_m^{1\prime}$ is "normal" (like that of a simple list) until E_{n+1}'' shifts to the right of E_k^1 (E_{n+1}'' differs from E_{n+1}' only by possible variable substitutions). The list looks like

$$E_1^1; ...; E_k^1; E_{n+1}''; E_{k+1}^{1\prime}; ...; E_m^{1\prime}$$

before computing the result of the interaction between E_k^1 and E_{n+1}''.

For argument's sake, let us assume that both E_{n+1}'' and E_k^1 bind variables to lists. They must then be of the form

x equaln headn cons tailn

and

x equalk headk cons tailk,

respectively, where "equaln" and "equalk" stand for equation signs. The resulting equations after rewriting become:

headk eq headn ; tailk eq tailn ; x equalk' headk cons tailk.

The entire list now looks like

$$E_1^1; ...; \{EQ;|TEMP;\} E_k^{1\prime}; ...$$

where {EQ;|TEMP;} is the sublist of newly-founded equations that were just generated in the last rewrite step, and $E_k^{1\prime}$ is

x equalk' headk cons tailk.

As with the base case, {EQ;|TEMP;} will become headed by a classified equation in a finite number of steps, if "false" does not occur (in which case termination occurs even sooner). The list then looks like:

$$x_1^1 = ...; ...; x_{k-1}^1 = ...; x_{new} = ...; \{EQ;|TEMP;\} x_k^1 = ...; ...; x_m^1 = ...$$

Now we have the case of a simple list of length k-1 attached by an equation (x sub new = ...). Since k-1 < n, by the induction hypothesis we have that the rewriting of the first k equations in the above list terminates in a finite number of steps. As with the base case, each newly-founded equation in the sublist {EQ;|TEMP;} will be absorbed by the sublist to its left in a finite number of steps. Since there are only a finite number of such equations, the entire list will be rewritten into:

$$x_1^2 = ...; ...; x_p^2 = ...; x_k^1 = ...; ...; x_m^1 = ...$$

in a finite number of steps.

Since $x_k^1, ..., x_m^1$ are distinct from $x_1^2, ..., x_p^2$ (since all occurrences of $x_k^1, ..., x_m^1$ in the other equations must have been substituted by the corresponding right hand sides already during interaction and shifting), the entire list is now a simple list. Therefore rewriting terminates by Theorem 6.1.

Earlier in the proof we assumed that E_k^1 and E_{n+1} ' are both binding a variable to a list. Similar arguments also apply to the case where E_k^1 and E_{n+1} ' are both binding a variable to a function. Therefore we have just proved Theorem 6.2.

Theorem 6.3. Any finite productive list (of equations) can be rewritten into a simple list in a finite number of steps.

Proof.
Let the productive list be

$$E_1; E_2; ...; E_n,$$

and let k ($0 < k < n$) be such that

$$E_1; E_2; ...; E_k$$

is a simple list and

$$E_1; E_2; ...; E_k; E_{k+1}$$

is productive. By Theorem 6.2,

$$E_1; E_2; ...; E_k; E_{k+1}$$

can be rewritten into a simple list in a finite number of steps. Let the resulting simple list be

$$E_1^1; ...; E_{k1}^1.$$

The complete list now looks like $E_1^1; ...; E_{k1}^1; E_{k+2}; ...; E_n$.

With an argument similar to the above, the list will become $E_1^2; ...; E_{k2}^2; E_{k+3}; ...; E_n$ in a finite number of steps, where $E_1^2; ...; E_{k2}^2$ is an irreducible list.

Since n is finite, eventually in a finite number of steps every E_{k+i} ($1 \leq i \leq n-k$) is absorbed and the list is transformed into a simple list.

Theorem 6.4 Any finite list of classified equations can be transformed into an irreducible list in a finite number of steps.

Proof.
Follows directly from Theorems 6.1 and 6.3. (Note that any list of classified equations is either simple or productive.)

Theorem 6.5 A newly-founded equation can be rewritten into an irreducible list in a finite number of steps.

Proof.
Consider first the newly-founded equation being in the form

term1 eq term2.

Let us look at the following two cases:

Case 1. "term1" and "term2" are both lists, or they are both general functions with the same function name, and they both have finite length. The result of rewriting this list would be

head1 eq head2; tail1 eq tail2
or
arglist1 temp arglist2

respectively, where "headi" is the head of the list "termi" (i = 1, 2) and "taili" the tail, and "arglisti" is the argument list of the general function "termi". The second equation list above ("arglist1 temp arglist2") becomes

arg11 eq arg21 ; ... ; arg1k eq arg2k

in a finite number of steps, where argij (i= 1,2; $1 \leq j \leq k$) is the jth parameter in arglisti, and k is the length of the parameter list (i.e., number of parameters) of arglisti. Of course, this is assuming that arglist1 and arglist2 have the same length, otherwise "false" occurs.

If one of "headi", "taili", and "argij" is again a list or general function, then decomposition continues. This process will terminate in a finite number of steps, for "term1" and "term2" have finite length, even if "false" does not occur. The list now looks like

term11 eq term21 ; ... ; term1m eq term2m,

where each termij (i= 1,2; $1 \leq j \leq m$) is a non-list and non-general function expression. In other words, each equation in the above list becomes non newly-founded in one rewrite step. Since non-newly-founded equations eventually become classified (since the Boolean equations are absorbed in a finite number of steps), rewriting has produced a finite classified equation list in a finite number of steps. From Theorem 6.4, rewriting terminates (unless "false" occurs in which case rewriting terminates even earlier).

Case 2. "term1" and "term2" are non-list and non-general function expressions. In one rewrite step "term1 eq term2" becomes non-newly-founded, and in zero or a finite number of steps it becomes classified. Rewriting terminates by Theorem 6.4.

Recall that originally we assumed the form

term1 eq term2

for the newly-founded equation in question. Now from the arguments in case 1, it is obvious that rewriting terminates as well for the other form of newly-founded equations:

term1 temp term2.

Theorem 6.6 For a finite list of newly-founded equations, rewriting terminates.

Proof.
Suppose the equation list is of the form

term11 equal term21; ... ; term1n equal term2n

where "equal" is either "eq" or "temp". Now we construct a new equation

newterm11 cons newterm12 cons ... cons newterm1n
newterm21 cons newterm22 cons ... cons newterm2n,

where newterm ij (i = 1, 2; $1 \leq j \leq n$) is:

1. termij if "term1j eq term2j" was present in the original list.
2. f func (termij) if "term1j temp term2j" was present where "f" is a general function name.

It is obvious that the original list can be derived from the newly constructed list in a finite number of steps by making equal the corresponding arguments of the newly constructed lists. From Theorem 6.5, rewriting therefore terminates.

Theorem 6.7 STAR is terminating.

Proof.
This is directly deduced from Theorem 6.6, since all input data are finite lists of newly-founded equations.

6.3. Confluence

Theorem 6.8 STAR is locally confluent.

Proof.
Recall that STAR has a hierarchical data structure that includes a set of operators each with a distinct precedence. This hierarchy guarantees that any arbitrary expression has a unique grouping, and therefore only one subexpression will have the highest precedence at any one time. Since Bertrand is designed such that the highest precedence subexpression is matched first, there will be no ambiguity (or alternative) for the rewriting of any given expression. In other words, STAR is locally confluent.

Theorem 6.9 STAR is confluent.

Proof.
Since STAR is both terminating and locally confluent, it is also confluent according to Bundy [Bund 83].

7. Results

In this chapter we present some complete unification examples to show how STAR works. We will first show one example without arithmetic, then a more complicated example with arithmetic. Next we show the factorial example to see how STAR fits in a complete logic programming frame (unification + control). And finally we show how we can run the factorial example backwards. We will show abbreviated traces for most of the examples.

For an explanation of the terminology below, see Figure 3 in Chapter 5.

7.1. A Simple Example - Without Arithmetic

The input to STAR is as follows:

```
main{
        "X" eq "Y";
        f func ("X") eq f func ("Y")
        f func (f func ("X")) eq "Y"
    }
```

The following is an abbreviated trace showing how unification works for the above example.

```
→  "X" eq "Y";
     f func ("X") eq f func ("Y");
     f func (f func ("X")) eq "Y"

→  "X" varsim "Y";
     f func ("X") eq f func ("Y");
     f func (f func ("X")) eq "Y"

→  "X" varsim "Y";
     "X" temp "Y";
     f func (f func ("X")) eq "Y"

→  "X" varsim "Y";
     "X" eq "Y";
     f func (f func ("X")) eq "Y"

→  "X" varsim "Y";
```

```
        "X" varsim "Y";
        f func (f func ("X")) eq "Y"

→       "X" varfin "Y";
        f func (f func("X")) eq "Y"

→       "X" varfin "Y";
        "Y" funsim  f func (f func("X"))

→       false
```

Note that the last rewriting step fails because both left hand sides of the equations occur in the right hand side of each other. This example shows that STAR has occur check capability.

7.2. A More Complicated Example

The example we will consider here was heavily tested. There are five equations in the input equation list and we have tested all of the 120 possible cases each having a distinct order for the input equations. All cases gave semantically equivalent, correct answers — although they do not look exactly the same, i.e., not syntactically identical.

The input for STAR looks like:

```
        main{
        "W" eq "Y" cons "V" cons "nil";
        "U" eq "Z" - 13;
        "U" + "Z" eq 17;
        "X" eq "Y" cons "nil";
        "W" eq "Z" + 5 cons "X" cons "nil"
        }
```

An abbreviated trace for the above input is shown as follows. Note that when we have "i,j" before a group of equations, we mean this group was derived from interaction of ith and jth equations in the previous group. Similar notes apply to all other numberings in the comments.

```
    →"W" eq "Y" cons "V" cons "nil";
        "U" eq "Z" - 13;
        "U" + "Z" eq 17;
        "X" eq "Y" cons "nil";
        "W" eq "Z" + 5 cons "X" cons "nil"

        (convert 1 and 2)
    →"W" lissim "Y" cons "V";
        "U" linsim 1** "Z" + + -13;
        "U" + "Z" eq 17;
        "X" eq "Y" cons "nil";
        "W" eq "Z" + 5 cons "X" cons "nil"

        (1,2 interaction twice)
    →"W" lissim "Y" cons "V";
```

```
   "U" linbar 1** "Z" + + -13;
   "U" + "Z" eq 17;
   "X" eq "Y" cons "nil";
   "W" eq "Z" + 5 cons "X" cons "nil"

   (convert 3)
→"W" lissim "Y" cons "V";
   "U" linbar 1** "Z" + + -13;
   "U" + "Z" - 17 = 0;
   "X" eq "Y" cons "nil";
   "W" eq "Z" + 5 cons "X" cons "nil"

   (1, 2)
→"W" lissim "Y" cons "V";
   1** "Z" + + -13 + "Z" - 17 = 0;
   "U" linbar 1** "Z" + + -13;
   "X" eq "Y" cons "nil";
   "W" eq "Z" + 5 cons "X" cons "nil"

   (1,2)
→1** "Z" + + -13 + "Z" - 17 = 0;
   "W" lisbar "Y" cons "V";
   "U" linbar 1** "Z" + + -13;
   "X" eq "Y" cons "nil";
   "W" eq "Z" + 5 cons "X" cons "nil"

   (reduce 1)
→"Z" numsim 15;
   "W" lisbar "Y" cons "V";
   "U" linbar 1** "Z" + + - 13;
   "X" eq "Y" cons "nil";
   "W" eq "Z" + 5 cons "X" cons "nil"

   (1,2)
→"W" lisbar "Y" cons "V" cons "nil";
   "Z" numbar 15;
   "U" linbar 1** "Z" + + - 13;
   "X" eq "Y" cons "nil";
   "W" eq "Z" + 5 cons "X" cons "nil"

   (2,3)
→"W" lisbar "Y" cons "V" cons "nil";
   "U" numsim 2;
   "Z" numfin 15;
   "X" eq "Y" cons "nil";
   "W" eq "Z" + 5 cons "X" cons "nil"

   (1, 2 interaction twice)
→"W" lisbar "Y" cons "V" cons "nil";
   "U" numbar 2;
   "Z" numfin 15;
```

```
    "X" eq "Y" cons "nil";
    "W" eq "Z" + 5 cons "X" cons "nil"

    (converted 4 interacts with 3)
→"W" lisbar "Y" cons "V" cons "nil";
    "U" numbar 2;
    "X" lissim "Y" cons "nil";
    "Z" numfin 15;
    "W" eq "Z" + 5 cons "X" cons "nil"

    (2,3)
→"W" lisbar "Y" cons "V" cons "nil";
    "X" lissim "Y" cons "nil";
    "U" numfin 2;
    "Z" numfin 15;
    "W" eq "Z" + 5 cons "X" cons "nil"

    (1,2)
→"X" lissim "Y" cons "nil";
    "W" lisfin "Y" cons "V" cons "nil";
    "U" numfin 2;
    "Z" numfin 15;
    "W" lissim "Z" + 5 cons "X" cons "nil"

    (4,5)
→"X" lissim "Y" cons "nil";
    "W" lisfin "Y" cons "V" cons "nil";
    "U" numfin 2;
    "W" lissim 20 cons "X" cons "nil";
    "Z" numfin 15

    (3,4)
→"X" lissim "Y" cons "nil";
    "W" lisfin "Y" cons "V" cons "nil";
    "W" lissim 20 cons "X" cons "nil";
    "U" numfin 2;
    "Z" numfin 15

    (2,3; note new equations are generated here)
→"X" lissim "Y" cons "nil";
    "Y" cons "V" cons "nil" eq 20 cons "X" cons "nil";
    "W" lisfin "Y" cons "V" cons "nil";
    "U" numfin 2;
    "Z" numfin 15

    (convert 2)
→"X" lissim "Y" cons "nil";
    "Y" eq 20;
    "V" cons "nil" eq "X" cons "nil";
    "W" lisfin "Y" cons "V" cons "nil";
    "U" numfin 2;
```

```
    "Z" numfin 15

    (1, 2 twice; convert 3)
→"X" gdsim 20 cons "nil";
    "Y" numfin 20;
    "V" eq "X";
    "W" lisfin "Y" cons "V" cons "nil";
    "U" numfin 2;
    "Z" numfin 15

    (2,3)
→"X" gdsim 20 cons "nil";
    "V" varsim "X";
    "Y" numfin 20;
    "W" lisfin "Y" cons "V" cons "nil";
    "U" numfin 2;
    "Z" numfin 15

    (1,2)
→"V" gdsim 20 cons "nil";
    "X" gdfin 20 cons "nil";
    "Y" numfin 20;
    "W" lisfin "Y" cons "V" cons "nil";
    "U" numfin 2;
    "Z" numfin 15

    (3,4)
→"V" gdsim 20 cons "nil";
    "X" gdfin 20 cons "nil";
    "W" lisfin 20 cons "V" cons "nil";
    "Y" numfin 20;
    "U" numfin 2;
    "Z" numfin 15

    (2,3)
→"V" gdsim 20 cons "nil";
    "W" lisfin 20 cons "V" cons "nil";
    "X" gdfin 20 cosn "nil";
    "Y" numfin 20;
    "U" numfin 2;
    "Z" numfin 15

    (1,2)
→"W" gdsim 20 cons (20 cons "nil") cons "nil";
    "V" gdfin 20 cons "nil";
    "X" gdfin 20 cosn "nil";
    "Y" numfin 20;
    "U" numfin 2;
    "Z" numfin 15
```

Note that traditional unification would fail for this example, since "U+Z" would not unify with 17. This example shows clearly how superior STAR is to traditional unification algorithms.

7.3. Factorial Example

The factorial function in Prolog is written as follows:

```
factorial(0, 1).
factorial(_n, _ans) :-
    _m is _n - 1,
    factorial(_m, _ans1),
    _ans is _n * _ans1.
```

If we try to write the second rule above in a more straightforward fashion, it does not work.

```
factorial(_n, _ans) :-
    factorial(_n - 1, _ans1),
    _ans is _n * _ans1.
```

If we ask "factorial (3, _ans)," the program fails to give an answer, because 3 does not unify with "_n - 1". For the same reason, further optimization is also impossible:

```
factorial(_n, _n * _ans1) :-
    factorial(_n - 1, _ans1).
```

STAR, however, with the addition of an appropriate control can easily handle the last optimization.

The following is a trace for the running of the query,

factorial(2, _U)

on the program

```
factorial(0, 1).
factorial(_n, _n * _x) :-
    factorial(_n - 1, _x).
```

Note that STAR accepts variables in the form of literals, so in the following we shall denote variables as capitalized literals.

Initially we try to match the query with the first rule of the program, and get the following (initial) input for STAR:

```
main{
2 eq 0;
"U" eq 1
}
```

STAR returns "false" ("false" occurs after converting "2 eq 0" into "2 = 0", which then is rewritten into "false").

The second rule, i.e., unify factorial(2, "U") with factorial(n, n * X), is now tried with input:

```
main{
2 eq "N1";
"U" eq "N1" * "X1"
}
```

Note that we use X1, X2, ..., and N1, N2, ... to represent variables that are involved because we keep all the variables from the beginning to the end.

STAR gives the following as a result:

"U" linsim 2** "X1" + + 0; "N1" numfin 2

The control now tries to run the body of the second rule, i.e., to match the body of the second rule with the first rule. The instantiated second rule follows:

factorial (N1, N1 * X1) :- factorial (N1 - 1, X1).

Therefore the following new equations are generated:

"N1" - 1 eq 0; "X1" eq 1

Attach these new equations to the resulting equation list from the last call, and we have the input equation list for our next call:

```
main{
"U" linsim 2 ** "X1" + + 0;
"N1" numfin 2;
"N1" - 1 eq 0;
"X1" eq 1
}
```

We attach new equations to the end of the existing list, because otherwise the interaction is incomplete. For example, if we have a list of equations as the output of STAR:

"X" numsim 4; "Y" numfin 5

and we have a new equation,

"Y" numsim 4,

attached to the beginning of the above list, hoping that the new list of equations

"Y" numsim 4; "X" numsim 4; "Y" numfin 5

will result in computation of a correct answer by the system. As the reader can see, after one rewriting step, we will get

"X" numsim 4; "Y" numfin 4; "Y" numfin 5

which will not further instantiate any rewrite rule. Therefore, the inconsistency between

"Y" numfin 4

and

"Y" numfin 5

will not be detected. This problem is easily solved, however, by simply attaching all new equations to the end of the old list of equations. This way, each and every equation in the old list will move rightward across each and every new equation, and the interaction will be complete.

Let us return to the factorial problem. The last call results in "false", for we have ""N1" numfin 2" and ""N1" - 1 eq 0".

Now we try the second rule, and get the following new equations:

"N1" - 1 eq "N2"; "X1" eq "N2" * "X2"

Attaching these to the end of the old list of equations, we have:

```
main{
"U" linsim 2 ** "X1" + + 0;
"N1" numfin 2;
"N1" - 1 eq "N2";
"X1" eq "N2" * "X2"
}
```

Running STAR with the above input, we get the result:

```
"U" linbar 2** "X2"+ + 0;
"X1" linfin 1** "X2"+ + 0;
"N2" numfin 1;
"N1" numfin 2
```

Next we try to match the body of the second rule with the first rule, and the second rule now looks like:

factorial("N2", "N2" * "X2") :- factorial("N2" - 1, "X2").

The new equations from this matching are:

"N2" - 1 eq 0; "X2" eq 1

which, when attached to the old list, make the new input:

```
main{
"U" linbar 2** "X2"+ + 0;
"X1" linfin 1** "X2"+ + 0;
"N2" numfin 1;
"N1" numfin 2;
"N2" - 1 eq 0;
"X2" eq 1
}
```

The resulting list of equations is as follows:

```
"X1" numsim 1;
"U" numfin 2;
"X2" numfin 1;
"N2" numfin 1;
"N1" numfin 2
```

Since all variables are now bound to numbers, we have finished the original query, factorial(2, "U"), and the answer is : "U" = 2. Unfortunately, STAR does not know which variables are intermediate and which are "desired," so we have to extract the final answer by hand. This is just an implementation detail, not part of the unification procedure. (See Figure 5 for an overview of the process.)

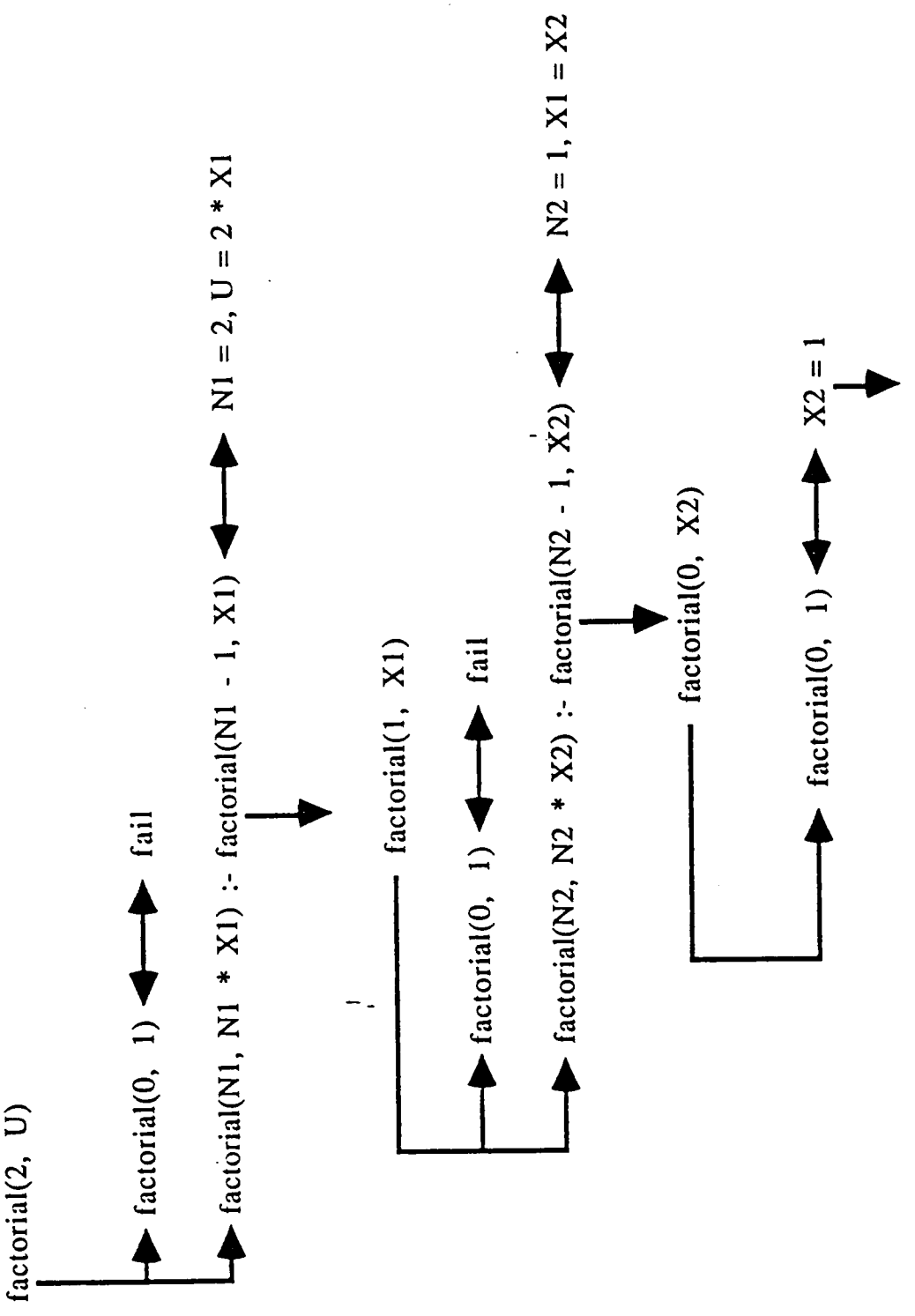

Figure 5. Running the factorial function forward

7.4. *Running Factorial Backward*

In this section we show a superb feature of STAR — how it can run the factorial function in reverse.

Traditional unification systems would fail when 2 (i.e., _ans) is unified with _n * 1 (i.e., _n * _ans1). With the arithmetic capability, however, STAR is able to reverse running of the factorial function.

To make it easy for the reader, we only run a simple query:

factorial(_N, 2).

With a similar process to that of Section 7.3, i.e., repeated unification with an incremented equation list, we eventually get the answer _N = 2. We will not show the trace here. See Figure 7.2 for an outline of how STAR computes the answer.

An example (backward factorial) that fails shows the limitations of our system. The query factorial(_W,5) sends the system [1] into an infinite loop. STAR generates the appropriate equations in answer to the queries, but there is always one more unknown variable than the number of equations. In addition, one of the original equations generated is non-linear; therefore as STAR generates new equations, the number of which is the same as the number of variables occurring in them, the non-linear equation never becomes linear, and STAR always returns a set of unsolved equations. Thus, the computation continues without getting any answer. In fact, the non-linear equation becomes more and more complicated as we generate more and more queries rather than becoming linear and solvable.

[1] STAR performs each unification; recursive calls are performed manually.

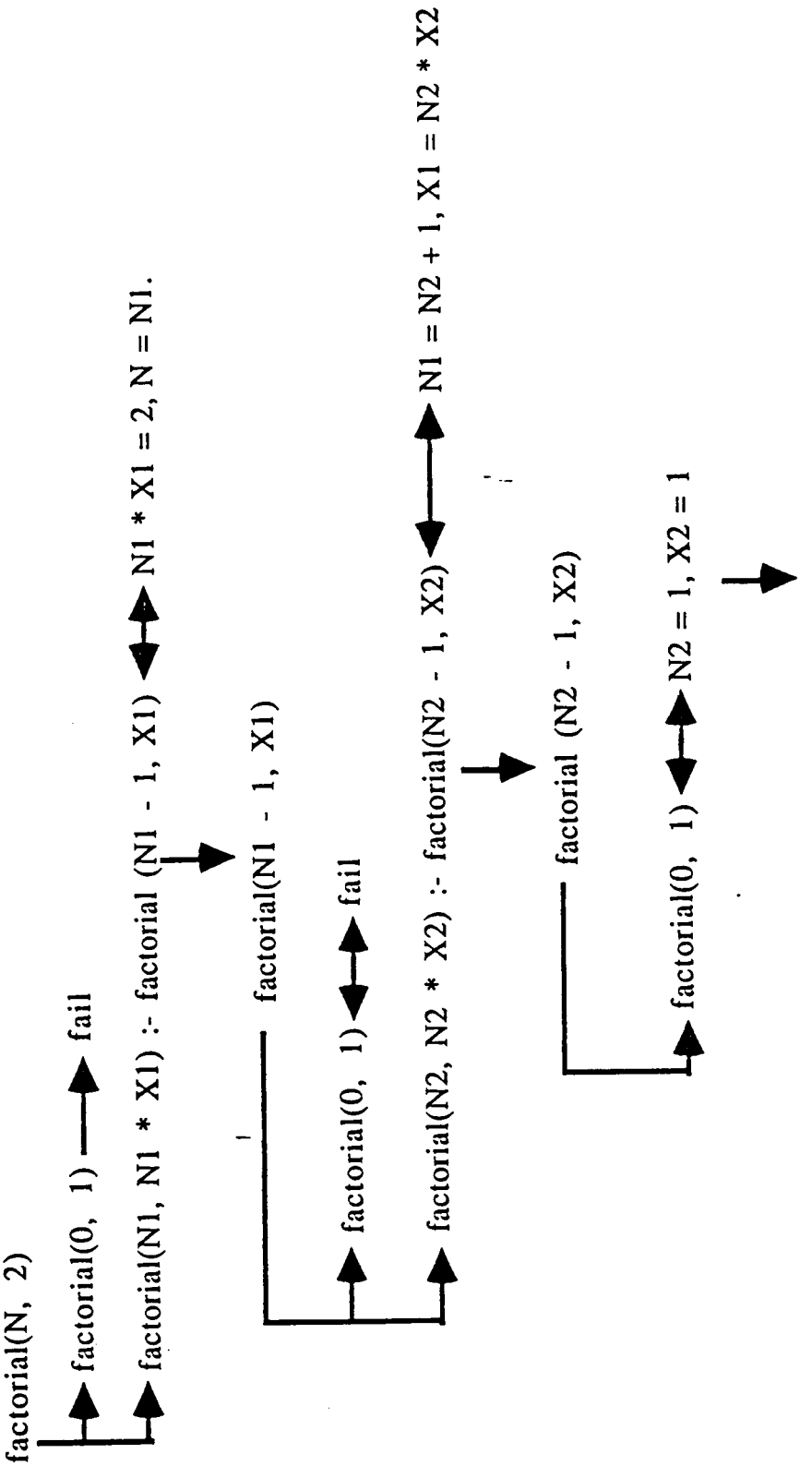

Figure 6. Running the factorial backwards

8. Conclusions

In this thesis we have developed a theory to incorporate function evaluation into unification by solving simultaneous equations. The theory is implemented using a stratified term rewriting system, STAR. We have therefore been able to create a system that unifies terms combining symbolic and numeric computations. STAR integrates symbolic and numeric computations into one theory at such a low level that in the implementation a uniform inference procedure solves the unification problem rather than requiring two independent procedures that call each other. We have achieved the ideal of treating simple arithmetic functions and purely symbolic computations (on trees, for example) as essentially the same process. We have shown that stratified term rewriting systems can be powerful enough to implement the core of logic programming languages.

With STAR, we can add new functions to the system with reasonable ease. To incorporate a new function into unification, we must be able to describe the semantics of the function as a relational specification (such as an equational theory). Then STAR code to solve expressions involving the new function must be written and integrated into the arithmetic unifier. The new code would work by implementing a constraint satisfaction system based on the relational semantics of the function specification. These requirements can be handled by STAR with relative ease due to its type structures and capability for constructing constraint satisfaction languages. Examples of possible such functions are combinations of the already defined numeric operations and inequalities.

In this thesis, we have demonstrated the superiority of stratified term rewriting techniques over narrowing and conditional term rewriting systems (let alone conventional term rewriting systems). Narrowing and conditional term rewriting are normally used to describe an equational theory only, rather than to implement (conventional) unification let alone to integrate the two. We have been able to integrate logic and functional programming in a way that none of the other systems have been able to achieve. Therefore STAR with its added capabilities appears to be more powerful than other term rewriting and narrowing systems. Stratified term rewriting techniques thus form a whole new field of study unto themselves. From a computational complexity point of view, STAR's non-linear equation reduction algorithm is inefficient (exponential in fact). But if most of the equations are reasonably simple, the actual cost should be much less than the worst case. If the programmer wants to solve many difficult non-linear equations, then STAR will run very slowly.

We note from an analysis of STAR's deficiencies some areas of future improvements and research. An obvious improvement is the generalization of the "is" operator so that we do not have to spend as much effort as we do now on binding the variables. An improvement specific to Bertrand would be the addition of a string comparison function

that would save a great amount of computational time; currently our system contains about 2500 (out of a total of 7500) rewrite rules just to compare strings. These two efficiency improvements should greatly speed up the algorithm although the availability of an appropriate "is" operator will violate the interaction rules comprising a large part of STAR. That is, we would need major changes to the algorithm should we have a workable "is" operator. Another promising research area would be to create a stratified term rewriting system that handles both control and unification - i.e., a complete equational logic programming implementation. We are already able to handle the unification part, so all that is left is formulation of the control component as a term rewriting system.

Bibliography

LNCS is an abbreviation for *Lecture Notes in Computer Science*, Springer-Verlag.

[AKNa 86] Hassan Ait-Kaci and Roger Nasr, "LOGIN: A Logic Programming Language with Built-In Inheritance," *J. Logic Programming*, Vol.3, No.3, pp.185-216.

[BaDe 86] L. Bachmair and N. Dershowitz,"Commutation, Transformation, and Termination," *Proceedings of the Eighth International Conference on Automated Deduction*, Oxford, England, *LNCS*, Vol.230, pp.5-20.

[BaPl 85] L. Bachmair and D.A. Plaisted,"Associative Path Ordering," *J. Symbolic Computation* Vol 1, pp.329-349.

[BeKl 82] J.A. Bergstra and J.W. Klop,"Conditional Rewrite Rules: Confluence and Termination," Report IW 198/82 MEI, Mathematische Centrum, Amsterdam.

[BrDa 78] D. Brand, J.A. Darringer, and W.H. Joyner, "Completeness of Conditional Reductions," Research Report RC 7404, IBM.

[Bund 83] Alan Bundy,*The Computer Modelling of Mathematical Reasoning*. Academic Press.

[ChKa 86] C. Choppy, S. Kaplan, and M. Soria, "Algorithmic Complexity of Term Rewriting Systems," *Rewriting Techniques and Applications, LNCS* Vol. 256, pp.256-273.

[Chur 41] A. Church, *The Calculi of Lambda Conversion*. Princeton University Press.

[Colm 82] Alain Colmerauer,"Prolog and Infinite Trees," *Logic Programming*, ed K.L. Clark and S.-A. Tarnlund, Academic Press.

[Colm 84] Alain Colmerauer, "Equations and Inequations on Finite and Infinite Trees," *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp.85-99, North-Holland.

[Dauc 88] Max Dauchet, "Termination of Rewriting Is Undecidable in the One-Rule Case," *Mathematical Foundations of Computer Science, LNCS* Vol.324, pp.262-270.

[Davi 87] Eanest Davis, "Constraint Propagation with Interval Labels," *Artificial Intelligence*, Vol. 32, pp. 281-331.

[DeFo 85] David Detlefs and Randy Forgaard,"A Procedure for Automatically Proving the Termination of a Set of Rewrite Rules," *Rewriting Techniques and Applications, LNCS*, Vol. 202, pp.255-270.

[DeMa 79] N. Dershowitz and Z. Manna, "Proving Termination with Multiset Orderings," *C. ACM*, Vol. 22, pp.465-476.

[DeOk 78] N. Dershowitz, Mitsuhiro Okada, and G. Sivakumar, "Canonical Conditional Rewrite Systems," *9th International Conference on Automated Deduction, LNCS* Vol.310, pp.538-549.

[Ders 81] N. Dershowita, "Termination of Linear Rewriting Systems," *Automata, Languages, and Programming, LNCS*, Vol.115, pp.448-458.

[Ders 84] N. Dershowitz,"Computing with Term Rewriting Systems," *Proceedings of an NSF Workshop on the Rewrite Rule Laboratory.*

[Ders 84b] N. Dershowitz,"Equations as Programming Language," *Fourth Jerusalem Conference on Information Technology*,pp.114 -123.

[Ders 87] N. Dershowitz,"Termination of Rewriting," *Journal of Symbolic Computation*, Vol.3, No.3, pp.69-115.

[Dros 83] K. Drosten,"Toward Executable Specifications Using Conditional Axioms," Report 83-01, t.U. Braunschweig.

[Fay 79] M. Fay, "First-Order Unification in an Equational Theory," *Proceedings of the 4th Workshop on Automated Deduction*, Austin, Texas pp.161-167.

[Gene 79] "Canonicality in Rule Systems," *Symbolic and Algebraic Computation, LNCS*,Vol.72, pp.23-29.

[GnLe 86] Isabelle Gnaedig and Pierre Lescanne, "Proving Termination of Associative Commutative Rewriting Systems by Rewriting," *Proceedings of 8th International Conference on Automated Deduction, LNCS*, Vol. 230, pp.52-61.

[GoMe 85] J. Goguen and J. Meseguer,"EQLOG: Equality, Types, and Generic Modules for Logic Programming," *Functional and Logic Programming*, ed. D. DeGroot and G. Lindstrom, Springer-Verlag.

[Gorn 67] S. Gorn,"Handling the Growth by Definition of Mechanical Languages," *Proceedings of the Spring Joint Computer Conference*,pp.213-224.

[Gorn 73] S. Gorn, "On the Conclusive Validation of Symbol Manipulation Processes," *J. Franklin Inst.*,Vol. 296, pp.499-518.

[HeRo 88] X. He, J.W. Roach, and R. Sundararajan,"A Systematic Study of Unification," VT Logic Programming Report #10.

[Huet 73] G.P. Huet, "The Undecidability of Unification in Third Order Logic," *Information and Control*, Vol.22, No.3.

[Hull 80] J.M. Hullot,"Canonical Forms and Unification," *Proceedings of the 5th Conference on Automated Deduction,LNCS*, Vol.87 pp.318-334, Springer-Verlag, Les Arcs, France.

[HuOp 80] G.P. Huet and D.C. Oppen,"Equations and Rewrite Rules - a Survey," *Formal Language Theory - Perspectives and Open Problems*, Academic Press.

[Itur 67] R. Iturriaga, "Contributions to Mechanical Mathematics," Ph.D. Thesis, Dept. Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

[JaLa 87] Joxan Jaffar and Jean-Louis Lassez,"Constraint Logic Programming," *Proceedings of the Conference on Principles of Programming Languages*, ACM, Munich.

[JoDe 86] A. Josephson and N. Dershowitz,"An Implementation of Narrowing: the RITE Way," *Proceedings of the Symposium on Logic Programming*, Salt Lake City, Utah.

[JoDe 89] Alan Josephson and Nachum Dershowitz, "An Implementation of Narrowing," *J. Logic Programming* 1989, pp.57-77.

[JoMu 84] J.-P. Jouannaud and M. Munoz,"Termination of a Set of Rules Modulo a Set of Equations," *Proceedings of the Seventh International Conference on Automated Deduction*, Napa, CA, *LNCS*, Vol. 170, pp.175-193.

[KaLe 80] S. Kamin and J.-J. Levy, "Two Generalizations of the Recursive Path Ordering," Unpublished note, Dept. Computer Science, U.Illinois, Urbana.

[KaNa 85] D. Kapur, P. Narendran, G. Sivakumar,"A Path Ordering for Proving Termination of Term Rewriting Systems," *Proceedings of the Tenth Colloquium on Trees in Algebra and Programming*, Berlin, West Germany, LNCS, Vol.185, pp.173-185.

[Kapl 84] S. Kaplan,"Fair Conditional Term Rewriting Systems: Unification, Termination, and Confluency," Laboratoire de Recherche en Informatique, Universite de Paris-Sud, Orsay, France.

[KnBe 70] D. Knuth and Bendix, "Simple Word Problems in Universal Algebras," *Computational Problems in Abstract Algebra*, ed. J. Leech, pp.163-279, Pergamon Press.

[Lank 75a] D.S. Lankford, "Canonical Algebraic Simplification in Computational Logic," Memo ATP-25, Automatic Theorem Proving Project, UT, Austin, Texas.

[Lank 75b] D.S. Lankford, "Canonical Inference," Memo ATP-32, Automatic Theorem Proving Project, UT, Austin, Texas.

[Lank 77] D.S. Lankford,"Some Approaches to Equality for Computational Logic: a Servey and Assessment," Memo ATP-36, Automatic Theorem Proving Project, UT Austin, Austin, Texas.

[Lank 79] D.S. Lankford,"Some New Approaches to the Theory and Applications of Conditional Term Rewriting Systems," Report MTP-6, Math Dept, Lousiana Tech U.

[Lank 79a] D.S. Lankford,"On Proving Term Rewriting Systems Are Noetherian," Memo MTP-3, Mathematics Dept, Louisiana Tech. University, Ruston, LA.

[Lass 87] Catherine Lassez,"Constraint Logic Programming," *BYTE*, August, pp.171-176.

[Lele 88a] Wm Leler,*Constraint Programming Languages*. Addison-Wesley Publishing Company.

[Lele 88b] Wm Leler,"Documentation for the Beta Release of Bertrand 88,"

[LiSn 77] R. Lipton and L. Snyder,"On the Halting of Tree Replacement Systems," *Proceedings of the Conference on Theoretical Computer Science*, U. Waterloo, Canada, pp.43-46.

[Lloy 84] J.W. Lloyd,*Foundations of Logic Programming*, Springer.

[MaNe 70] Z. Manna and S. Ness,"On the Termination of Markov Algorithms," *Proceedings of the Third Hawaii International Conference on System Science*, Honolulu, HI, pp.789-792.

[Pett 81] A. Pettorossi, "Comparing and Putting Together Recursive Path Orderings, Simplification Orderings and Non-Ascending Property for Termination Proofs of Term Rewriting Systems," *Proceedings of the Eighth EATCS International Colloquium on Automata, Languages and Programming*, Acre, Israel, *LNCS*, Vol.115, pp.432-447.

[Plai 78a] D.A. Plaisted,"Well-founded Orderings for Proving Termination of Systems of Rewrite Rules," Report R-78-932, Dept Computer Science, U. Illinois, Urbana.

[Plai 78b] D.A. Plaisted,"A Recursively Defined Ordering for Proving Termination of Term Rewriting Systems," Report R-78-943, Dept. Computer Science, U. Illinois, Urbana.

[PlEn 82] U. Pletat, G. Engels, and H.D. Ehrich,"Operational Semantics of Algebraic Specifications with Conditional Equations," *7eme CAAP*, Lille,*LNCS*.

[ReKi 85] P. Rety, C. Kirchner, H. Kirchner, and P. Lescanne, "Narrower: a New Algorithm for Unification and Its Application to Logic Programming," *Proceedings of the 1st Conference on Rewriting Techniques and Applications, LNCS*, Vol.202 pp.141-157, Dijon, France.

[Remy 83] J.L. Remy,"Proving Conditional Identities by Equational Case Reasoning Rewriting and Normalization," *Actes Seminaire Laboratoire Informatique Theorique de Paris*.

[Rety 87] Pierre Rety, "Improving Basic Narrowing Techniques," *Rewriting Techniques and Applications, LNCS,* Vol.256, pp.228-241.

[Robi 65] J.A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *JACM,* Vol.12, No.1, pp.23-41.

[RoSu 89] J.W. Roach, R. Sundararajan, and L.T. Watson, "Replacing Unification by Constraint Satisfaction to Improve Logic Programming Expressiveness," *J. Automated Reasoning,* to be published.

[Rusi 87] M. Rusinowitch,"Plaisted Ordering and Recursive Decomposition Ordering Revisited," *J. Symbolic Computation,* Vol.3,No.3.

[Slag 74] J.R. Slagle,"Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity," *J. ACM,* Vol.21,pp.622-642.

[Stic 81] M.E. Stickel, "A Unification Algorithm for Associative-Commutative Functions," *JACM,* Vol.28, No.3.

[YoSu 86] J. You and P.A. Subrahmanyam, "E-Unification Algorithms for a Class of Confluent Term Rewriting Systems," *Automata, Languages and Programming, LNCS* Vol.226, pp.454-463.

[Zhen 86] B. Zheng,"Adapting Colmerauer's Tree Rewriting Algorithm to Structure Sharing," VT Logic Programming Report #13, Blacksburg, VA.

[ZhRe 85] Hantao Zhang and Jean-Luc Remy, "Contextual Rewriting," *Rewriting Techniques and Applications, LNCS* Vol.202, Dijon, France.

The vita has been removed from the scanned document