PROOFS AND COMPUTATIONS IN CONDITIONAL EQUATIONAL THEORIES

BY

G. SIVAKUMAR

B.Tech., Indian Institute of Technology, 1982
M.S., Rensselaer Polytechnic Institute, 1984

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1989

Urbana, Illinois

PROOFS AND COMPUTATIONS
IN CONDITIONAL EQUATIONAL THEORIES

G. Sivakumar, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1989
Nachum Dershowitz, Advisor

Conditional equations arise naturally in the algebraic specification of data types. They also provide an elegant computational paradigm that cleanly combines logic and functional programming. In this thesis, we study how to do proofs and computations in conditional equational theories, using rewriting techniques.

We examine different formulations of conditional equations as rewrite systems and compare their expressive power. We identify a class of "decreasing" systems for which most of the basic notions (like rewriting and computing normal forms) are decidable. We then study how to determine if a conditional rewrite system is "confluent." We settle negatively the question whether "joinability of critical pairs" is, in general, sufficient for confluence of terminating conditional systems. We also prove two positive results for systems having critical pairs and arbitrarily big terms in conditions.

We discuss "completion" methods to generate convergent conditional rewrite systems equivalent to a given set of conditional equations. Finally, we study equation solving methods and formulate a goal-directed approach that improves prior methods and detects more unsatisfiable equations.

To my parents.

ACKNOWLEDGEMENTS

I sincerely thank Nachum Dershowitz, my thesis advisor, for his constant encouragement, guidance and the confidence he has placed in me. I will always remember his gracious hospitality and help, when I visited him at the Hebrew University in January 1988. Most of the results reported here evolved in discussions and correspondence with him. His comments have also helped me immensely in improving the presentation.

I also thank Mitsuhiro Okada for his collaboration, and the many fruitful discussions we had while developing the results on confluence presented in Chapters 3 and 4. The counter-example presented in Chapter 4 is a simplified version of his original system.

I would like to express my gratitude to Deepak Kapur, who first introduced me to this research area, for his encouragement and advice. I am also grateful to Uday Reddy for his comments and the many discussions we had.

Finally, I thank all my friends in Urbana for all the fun and dinners they provided, and my parents for their patience and understanding.

TABLE OF CONTENTS

1 INTRODUCTION 1
1.1 Motivation 1
1.2 Structure of the Thesis 4

2 UNCONDITIONAL EQUATIONS AND REWRITE SYSTEMS 6
2.1 Basic Syntax 6
2.2 Equational Theories 8
2.3 Rewrite Systems 9

3 CONDITIONAL EQUATIONS AND REWRITE SYSTEMS 12
3.1 Conditional Equations 12
3.2 Some Translations of Conditional Equations 15
3.2.1 Conservative Extensions 15
3.2.2 Translating to Unconditional Equations 16
3.3 Conditional Rewrite Systems 17
3.4 Termination of Conditional Rewriting 21
3.5 Strength of Rewrite Systems 21
3.6 Decreasing Systems 24

4 CONFLUENCE OF CONDITIONAL REWRITE SYSTEMS 28
4.1 Local Confluence and Critical Pairs 28
4.1.1 Disjoint Peaks 29
4.1.2 Variable Peaks 29
4.1.3 Critical Peaks and Conditional Critical Pairs 30
4.2 Counter-Examples 32
4.3 Confluence of Decreasing Systems 35
4.4 Confluence of Left-Linear, Normal Systems 39
4.5 Confluence of Overlay Systems 41
4.6 Conclusion 46

5 COMPLETION METHODS FOR CONDITIONAL EQUATIONS 47
5.1 Unconditional Completion 48
5.2 Conditional Completion Methods 51
5.2.1 Handling Non-Decreasing Equations 53
5.2.2 Contextual Simplification 56
5.2.3 Critical Pair Optimization 59
5.2.4 An Interesting Example 61
5.3 Conclusion 64

6 EQUATION SOLVING METHODS . . . 65
6.1 Equational Programming . . . 65
6.2 Procedures for Equation Solving . . . 67
6.2.1 The Narrowing Procedure . . . 69
6.2.2 Drawbacks of Narrowing . . . 70
6.2.3 The Decomposition Procedure . . . 72
6.2.4 Drawbacks of Decomposition . . . 73
6.3 Goal Directed Equation Solving . . . 75
6.3.1 Innermost Rewriting Sequences and Oriented Goals . . . 75
6.3.2 Simulating Innermost Rewriting . . . 76
6.3.3 Operator Rewriting . . . 78
6.4 Completeness of Equation Solving Procedures . . . 80

7 SUMMARY AND FUTURE WORK . . . 82

APPENDIX A . . . 84

APPENDIX B . . . 90

REFERENCES . . . 100

VITA . . . 105

1 INTRODUCTION

1.1 Motivation

Equational reasoning is very important in many areas of computer science, like symbolic algebraic computation, automated theorem proving, program specification and verification, and high-level (logic and functional) programming languages. An equational theory is specified by a set of axioms of the form $s = t$ (where s and t are terms, perhaps containing variables); the "theory" being the set of equations inferable by "replacing equals with equals." For example, the equations below specify the identity, inverse and associative properties of a group:

$$\begin{array}{rcl} 0 + x & = & x \\ -(x) + x & = & 0 \\ (x + y) + z & = & x + (y + z) \end{array}$$

The *validity* (or *word*) problem is that of determining if an identity follows logically from the given axioms. For example, we may ask if $-0 = 0$ is a valid consequence of the group axioms above, i.e. if it is true in all models of the axioms. The *satisfiability* (or equation solving) problem is that of finding substitutions for the variables that make two terms equal in all models. For example, the equation $-x = x$ is satisfied by the substitution $\{x \mapsto 0\}$.

Efficient methods for handling the validity and satisfiability problems for equational theories have been studied for some time now, and are the focus of most of the work in the area of *rewrite systems.* Rewrite systems are collections of *directed* equations (rules) of the form $l \rightarrow r$. Rules can be used to *simplify* a term by repeatedly replacing instances of left-hand sides l by the corresponding

instance of the right-hand side r (and not the other way) until a simplest possible (*normal*) form is obtained.

Rewrite systems possessing nice properties like *termination* and *confluence*, and equivalent to a given equational theory, may be generated by *completion* methods. With such systems, the word problem can be decided by simply checking if the two terms have the same normal form. In addition to producing efficient decision procedures for some equational theories, completion-like methods have also been suggested as a replacement for paramodulation in resolution-based theorem provers that handle equality [Lankford, 1975]. For (refutational) theorem proving in first-order predicate calculus, Hsiang [Hsiang, 1982] showed how a variant of completion can be used in place of resolution. Rewriting methods have also been used to prove inductive theorems [Musser, 1980] by showing that the hypothesis can cause no inconsistency. Thus, in many applications, rewriting methods have turned out to be a very successful approach to equational reasoning. For a comprehensive survey of the work in this field see [Dershowitz and Jouannaud, 1989].

Most of the work on rewrite systems has dealt only with pure or unconditional equations. A *conditional equation* is an equational implication of the form:

$$p_1 = q_1 \wedge \cdots \wedge p_n = q_n : s = t$$

for $n \geq 0$ (that is, a universal Horn clause with equality literals only). Unlike an unconditional equation $s = t$, which says that for all substitutions for variables in s and t, the two terms are equal, here they are equal only for those substitutions for which the condition $p_1 = q_1 \wedge \cdots \wedge p_n = q_n$ "holds." An example of a conditional equation with only one premise is:

$$x > 0 = true : factorial(x) = x * factorial(x - 1)$$

Conditional equations are very useful in the algebraic specification of abstract data types (since initial algebras exist). They have been studied largely from this perspective in [Bergstra and Klop, 1982], [Kaplan, 1984] and [Zhang and Rémy, 1985]. The equations below, for example, define some operations on stacks.

$$\begin{array}{rcl} top(push(x,y)) & = & x \\ pop(push(x,y)) & = & y \\ empty?(x) = false \;:\; push(top(x),pop(x)) & = & x \end{array}$$

Checking whether such specifications are consistent or complete requires methods for handling the word problem for conditional equational theories.

The application of conditional equations as a programming language has been proposed in [Dershowitz and Plaisted, 1985], [Fribourg, 1985] and [Goguen and Meseguer, 1986]. A program is a set of conditional equations, and a computation attempts to find a substitution that makes two terms provably equal in the underlying theory. This paradigm integrates cleanly the logic programming ability of Prolog with built-in equality and the functional capability of Lisp. An interpreter for such a language can be viewed as an equation solver for conditional equations.

The word problem and satisfiability problem for conditional equations can be shown semi-decidable by brute-force enumeration methods. Analogous to unconditional equations, it is natural to ask if we can get more efficient methods using *conditional rewriting*. This is the main focus of this thesis. We study how to do proofs and computations in conditional equational theories using conditional rewriting. Many of the concepts and techniques of unconditional rewriting carry over to conditional systems. But, corresponding results about confluence of conditional rewrite systems and completion methods for conditional equations have been hard to obtain without very strong restrictions on the class of allowable rules.

We first examine different formulations of conditional rewriting. We show how to extend confluence results for conditional rewriting in a more general framework. We develop a formulation of a completion procedure for conditional equations, similar to that proposed in [Ganzinger, 1987], that can handle many common examples. We also address the satisfiability problem (solving equations) in conditional theories and formulate a "goal directed" equation solving method that improves prior techniques and can serve as the basis for interpreters of equational languages.

1.2 Structure of the Thesis

The background material in Chapter 2 gives an overview of unconditional equational theories and rewrite systems. It introduces most of the terminology and concepts used in the rest of the thesis. Readers familiar with this area need only skim through this.

In Chapter 3, conditional equational theories are described. We examine different formulations of conditional equations as rewrite systems and compare their expressive power. We then examine a restriction of these systems using a "decreasing" ordering. With this restriction, most of the basic notions (like rewriting and computing normal forms) are decidable.

In Chapter 4, we study the confluence of conditional rewriting in detail. We settle negatively the question whether "joinability of critical pairs" is, in general, sufficient for confluence of terminating conditional systems. We review known sufficient conditions for confluence, and also prove two new positive results for systems having critical pairs and arbitrarily large terms in conditions.

In Chapter 5, a completion method is proposed to generate convergent conditional rewrite systems equivalent to a given set of conditional equations. Techniques are given to handle non-decreasing equations and critical pairs, by converting them to equivalent unconditional equations, using a conservative extension of the theory. We give several examples to show how this works.

We first briefly review, in Chapter 6, the use of conditional equations as a programming language. We then study equation solving techniques and examine methods to improve them. We formulate a goal-directed equation solving technique, that captures the features of narrowing and top-down decomposition.

We conclude with a summary and directions for further work in Chapter 7. In Appendix A, we briefly describe a preliminary implementation of conditional completion in RRL—a rewrite rule laboratory—and give an example. In Appendix B, we give a Prolog implementation of equation solving and a transcript of some examples.

2 UNCONDITIONAL EQUATIONS AND REWRITE SYSTEMS

In this chapter, we briefly review the basic notions and results for unconditional equational theories. Most of the terminology used in the rest of the thesis is explained here. A comprehensive survey of this area is [Dershowitz and Jouannaud, 1989]. Other surveys are [Huet and Oppen, 1980], [Klop, 1986].

2.1 Basic Syntax

We work with a set $T(F, X)$ of terms constructed from a (countable) set F of function symbols and a (countable) set X of variables. We use just T for the set of terms when F and X are clear from the context. We normally use the letters a through h for function symbols; l, r, and p through w for arbitrary terms; x, y, and z for variables.

Each function symbol $f \in F$ has an *arity* $n \geq 0$ which is the number of arguments (immediate subterms) that it has in a well-formed term. *Constants* are function symbols of arity zero. Variable-free terms are called *ground*. The set $G(F)$ of ground terms is, therefore, $T(F, \emptyset)$.

A term t in $T(F, X)$ may be viewed as a finite ordered tree. Internal nodes are labeled with function symbols (from F) of arity greater than 0. The outdegree of an internal node is the same as the arity of the label. Leaves are labeled with either variables (from X) or constants.

We use $u[t]$ to denote a term that has t as a subterm. We use $u[\cdot]$ to denote the *context* in which t occurs in a term $u[t]$. The context is the tree obtained by deleting t from the tree. By $t\mid_\pi$, we denote the *subterm* of t rooted at *position* π. A subterm of t is called *proper* if it is distinct from t.

Positions can, for example, be represented in Dewey decimal notation (a sequence of positive integers, describing the path from the outermost, "root" symbol to the head of the subterm at that

position). Thus $f(g(a), h(b))\,|_{1.1}$ is the subterm a and $f(g(a), h(b))\,|_{2.1}$ is the subterm b. By $t[s]_\pi$ we denote the term obtained from t by replacing the subterm at position π by the term s. For example, if $t = f(x, y)$ and $\pi = 1$, then $t[a]_\pi$ is the term $f(a, y)$. Position π_1 is said to be *above* position π_2, if π_1 is a proper prefix of π_2. In this case, $t\,|_{\pi_2}$ is a proper subterm of $t\,|_{\pi_1}$; we also say that π_2 is *below* π_1. Positions π_1 and π_2 are *independent* positions if neither one is a prefix of the other; the subterms $t\,|_{\pi_1}$ and $t\,|_{\pi_2}$ are said to be *disjoint*.

A *substitution* is a mapping from variables to terms. It is usually an identity function on all but finitely many variables. We use lower case Greek letters for substitutions, and write it out as $\{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$. A substitution σ can be extended to a function from the set of terms T to itself. A *composition* of two substitutions, denoted by juxtaposition, is just the composition of the two functions. We say that a substitution σ is *at least as general* as a substitution ρ if there exists a substitution τ such that $\sigma\tau = \rho$.

A term t *matches* a term s if $t = s\sigma$ for some substitution σ. We also say that t is an *instance* of s in this case. A term s *unifies* with a term t if $t\sigma = s\sigma$ for some substitution σ.

We use $\rightarrow$ (sometimes with subscripts) to denote a binary relation over a set of terms. A relation $\rightarrow$ is called a *rewrite* relation (or "monotonic") if $s \rightarrow t$ implies that $u[s\sigma]_\pi \rightarrow u[t\sigma]_\pi$, for all contexts $u[\cdot]$, terms s and t, positions π, and substitutions σ. If $\rightarrow$ is a binary relation on T, then by $\leftarrow$ we denote its inverse, by $\leftrightarrow$ its symmetric closure, by $\rightarrow^=$ its reflexive closure, by $\rightarrow^+$ its transitive closure, and by $\rightarrow^*$ its reflexive-transitive closure.

A binary relation $\rightarrow$ on a set T is said to be *terminating* if there exists no endless chain $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \cdots$ of elements of T, i.e. if its transitive closure $\rightarrow^+$ is well-founded. Terminating relations are useful for doing inductive proofs.

2.2 Equational Theories

An *equation* is an *unordered* pair of terms, written in the form $s = t$. Either or both of s and t may contain variables; which are understood as being universally quantified. A (finite or infinite) set of equations E, for example, the group axioms below:

$$\begin{array}{rcl} 0 + x & = & x \\ -(x) + x & = & 0 \\ (x + y) + z & = & x + (y + z) \end{array}$$

specifies an *equational theory* $\underset{E}{=}$, over the set of terms T, that is obtained by taking reflexivity, symmetry, transitivity, and context application as inference rules and all instances of equations in E as axioms.

By the completeness of first order predicate calculus with equality, validity and provability coincide. So, we can also define the equational theory using a *replacement* relation $\leftrightarrow$, based on the idea of "replacement of equals for equals." We write $s \leftrightarrow t$, for terms s and t in T, if $l = r$ is an equation in E, $s\,|_{\pi}= l\sigma$ and $t = s[r\sigma]_{\pi}$. Intuitively, we can replace an instance of one side of an equation in E by the corresponding instance of the other side of the equation. For example, if $0 + x = x$ is an equation in E, then $f(0 + 0) \leftrightarrow f(0)$. The substitution $\{x \mapsto 0\}$ is used in this replacement.

It can be shown that $s \underset{E}{=} t$ iff $s \leftrightarrow^* t$, where $\leftrightarrow^*$ is the reflexive-transitive closure of $\leftrightarrow$. In other words, two terms are provably equal if one may be obtained from the other by a finite number of replacements of equal subterms. An equational *proof* of $s = t$, is, therefore, a sequence of such replacement steps–

$$s = s_0 \leftrightarrow s_1 \cdots \leftrightarrow s_n = t$$

of $n \geq 0$ applications of equational axioms.

The *word problem* for a set of equations E, is the question whether an equation $s = t$ between two *ground* terms, s and t follows from E, i.e., is $s \leftrightarrow^* t$? For example, we may ask if $-0 = 0$ is a consequence of the group axioms. When s and t are not ground terms, checking if $s \leftrightarrow^* t$ is referred to as the *validity problem.*

The *satisifiability* problem is the question whether there exists a substitution σ for variables in two terms s and t such that $s\sigma \leftrightarrow^* t\sigma$. For example, the equation $-x = x$ is satisfied by the substitution $\{x \mapsto 0\}$.

2.3 Rewrite Systems

A rewrite *rule* over a set of terms T is an ordered pair (l, r) of terms, and is written $l \rightarrow r$. If no variable occurs more than once in l, then the rule is said to be *left-linear.* Similarly, a rule is *right-linear* if no variable is repeated in r, and is *linear* if it is both left-linear and right-linear. A *rewrite system* (or *term rewriting system*) R is a (finite or infinite) set of such rules. Rules can be used to replace instances of l by corresponding instances of r; but unlike equations, they cannot be used in the reverse direction (that is, to replace instances of the right-hand side r). This is, in fact, the main idea of rewriting– to impose directionality on the use of equations.

We use $\rightarrow$ to denote the rewrite relation. We say that a term s in T *rewrites* to a term t in T, denoted $s \rightarrow t$, if $s\mid_\pi = l\sigma$ and $t = s[r\sigma]_\pi$, for some rule $l \rightarrow r$ in R, position π in s, and substitution σ. We say that t is *derivable* from s if $s \rightarrow^* t$, where $\rightarrow^*$ is the reflexive-transitive closure (zero or more steps) of $\rightarrow$.

A term s is *reducible* by R if there is a term t such that $s \rightarrow t$; otherwise we say that s is *irreducible* or in *normal* form. We write $s \xrightarrow{!} t$ if $s \rightarrow^* t$ and t is irreducible, in which case we say

that t is a normal form of s. Two terms s and t are said to be *joinable* written $(s \downarrow t)$ if there is a term u which is derivable from both s and t, i.e. such that $s \rightarrow^* u \; {}^*\!\!\leftarrow t$.

A rewriting system R is *terminating* for a set of terms T if the rewrite relation $\rightarrow$ over T is terminating. That is, there is no infinite sequence of terms t_i in T such that $t \rightarrow t_1 \rightarrow t_2 \cdots$. When a system is terminating, every term has at least one normal form. Note that a terminating system cannot have any rule, like $-x + x \rightarrow -y + y$, with a variable on the right that is not also on the left (since y could, for example, be $-x + x$), nor can a left-hand side be just a variable, like $x \rightarrow 0 + x$. A comprehensive survey of methods for establishing termination is [Dershowitz, 1987].

A rewrite relation is *confluent* if whenever two terms, s and t, are derivable from a term u, then a term v is derivable from both s and t. That is, if $u \rightarrow^* s$ and $u \rightarrow^* t$, then there is a term v such that $s \rightarrow^* v$ and $t \rightarrow^* v$. Confluence says that if two terms have a common ancestor, they also have a common descendent. *Ground confluence* is confluence restricted to ground terms. That is, a system is ground confluent whenever any two terms that are derivable from a ground term are joinable.

A system R is *convergent* if it is both terminating and confluent. A convergent system has the *unique normalization* property. In other words, every term t in T possesses exactly one normal form. This means that terms s and t are joinable by a convergent system R iff they have the same normal form. A system that is convergent over ground terms is said to be *ground convergent.*

Let $l \rightarrow r$ and $g \rightarrow d$ be two rules (or two versions of the same rule–i.e. with variables renamed) in R. The equation $s = t$ is said to be a *critical pair* between these two rules, if g unifies with a non-variable subterm of l at position π using a substitution σ and $s = r\sigma$ and $t = l[d]_\pi\sigma$. We say that $l\sigma$ is a critical *overlap*, and we have $r\sigma \;\leftarrow\; l\sigma \;\rightarrow\; l[d]_\pi\sigma$. That is, s and t are the two terms we obtain by rewriting the overlap between the two rules.

For example, $0 + (u + v) = u + v$ is a (joinable) critical pair between the rules $0 + x \rightarrow x$ and $(y + u) + v \rightarrow y + (u + v)$. The overlap between these two rules is $(0 + u) + v$; it is obtained by unifying a left-hand side $0 + x$ with a non-variable subterm $y + u$ of the other left-hand side. A critical pair $s = t$ in R is joinable if $s \downarrow t$ in R.

Critical pairs are useful for checking if a terminating system is confluent, hence convergent. We have the following famous lemma:

Lemma 2.1 (Critical Pair Lemma [Knuth and Bendix, 1970]**)** *A terminating rewrite system is convergent iff all its critical pairs are joinable.*

A rewrite system R is *sound* with respect to a of a set of equations E, if the derivability relation $\rightarrow^*$ of R is a subset of the replacement relation $\leftrightarrow^*$ of E. That is for any two terms, s and t, $s \rightarrow^* t$ using R only if $s \leftrightarrow^* t$ in E. A system R is *complete* for E, if any two terms that are provably equal in E are joinable in R. That is, $s \downarrow t$ in R whenever $s \leftrightarrow^* t$ in E.

If R is both sound and complete for E, then the validity problem for E—is $s \leftrightarrow^* t$?—is the same as checking if s and t are joinable in R. If R is also finite and convergent, then this can be done quite efficiently by just checking if s and t have the same normal form. Thus, for those equational theories for which we can find finite, convergent rewrite systems that are sound and complete, we have an effective decision procedure for the validity problem.

Completion methods can be used to generate such systems from given equational axioms. We discuss this in Chapter 5. In Chapter 6, we will see how convergent rewrite systems also provide more efficient methods for the satisfiability problem in equational theories.

3 CONDITIONAL EQUATIONS AND REWRITE SYSTEMS

In this chapter, we first describe conditional equational theories. Then, we study different ways of formulating them as conditional rewrite rules and compare their expressive power. We identify a class of decreasing systems for which most of the interesting notions are decidable. Decreasing systems have somewhat weaker restrictions than the simplification systems or reductive systems studied previously in [Kaplan, 1984] and [Jouannaud and Waldmann, 1986]. For this class of systems, we also show that various formulations as rewrite systems are in some sense equivalent. In the next chapter, we will see how decreasing systems also satisfy the critical pair lemma for confluence and that straightforward attempts at further weakening these restrictions do not work.

3.1 Conditional Equations

A (positive) *conditional equation* takes the form:

$$p_1 \;=\; q_1 \;\wedge\; \cdots \;\wedge\; p_n \;=\; q_n \;:\; s \;=\; t$$

where $n \geq 0$. The $p_i = q_i$ are equations, possibly containing (universally quantified) variables. An example of a conditional equation with only one premise is:

$$x > 0 \;=\; \mathit{true} \;:\; \mathit{factorial}(x) \;=\; x * \mathit{factorial}(x-1)$$

The ":" may be thought of as implication, with $s = t$ as the conclusion and $p_i = q_i$ as the premises. We will sometimes write $c : s = t$ to denote a conditional equation where it is understood that c denotes equations $p_1 \;=\; q_1 \;\wedge\; \cdots \;\wedge\; p_n \;=\; q_n$.

Let E be a set of conditional equations. We define the one-step replacement relation $\leftrightarrow$ and its reflexive-transitive closure $\leftrightarrow^*$ as follows. If

$$p_1 = q_1 \wedge \cdots \wedge p_n = q_n : s = t$$

is a conditional equation, σ is a substitution, $u[s\sigma]$ is a term with $s\sigma$ as a subterm at position π, and $p_i\sigma \leftrightarrow^* q_i\sigma$ for $i = 1, ..., n$, then $u[s\sigma] \leftrightarrow u[t\sigma]_\pi$ where $u[t\sigma]_\pi$ is the term obtained by replacing $s\sigma$ by $t\sigma$.

Intuitively, only for substitutions that are *feasible* (for which the conditions can be proved recursively by a sequence of such replacements), can we replace that instance of one side of a conditional equation by the corresponding instance of the other side. An unconditional equation $s = t$, however, applies for all substitutions. We write $E \vdash s = t$ if $s \leftrightarrow^* t$ for a set E of conditional equations.

For example, using equations:

$$\begin{array}{rcl} & 0 + y & = & y \\ x + y = z \; : & s(x) + y & = & z \end{array}$$

we have $s(0) + s(0) \leftrightarrow s(s(0))$ using the second equation, since for the substitution $\{x \mapsto 0, y \mapsto s(0), z \mapsto s(0)\}$ the condition $x + y = z$ is feasible (as $0 + s(0) \leftrightarrow s(0)$ using the first (unconditional) equation).

Note that we do not restrict terms in the conditions to contain only variables present in the equation in the conclusion. Variables are universally quantified as, for example, below:

$$\forall \vec{x}, \vec{y}\; [c(\vec{x}, \vec{y}) \; : \; s(\vec{x}) = t(\vec{x})]$$

where $\vec{y}$ are those ("extra") variables in the terms in the condition that do not also appear in either side of the conclusion. This is equivalent to

$$\forall \vec{x}\ [(\exists \vec{y}\ c(\vec{x},\vec{y})):\ s(\vec{x}) = t(\vec{x})]$$

Operationally, this means that, when there variables in the condition not present in either side of the equation in the conclusion, we have to find a feasible substitution for the extra variables before we can replace $s\sigma$ by $t\sigma$ in any context.

Though we sometimes use uninterpreted constants like *true* and *false* in conditions, we will consider only *equational* (algebraic) consequences and proofs in this thesis. Note that equational logic lacks a "law of excluded middle." That is, we cannot conclude that $f(x) = 0$ from the equations:

$$\begin{array}{rcl} p(x) = true : f(x) & = & 0 \\ p(x) = false : f(x) & = & 0 \end{array}$$

by reasoning that $p(x)$ must be either *true* or *false*. We have to prove (equationally) that the condition holds. The relation between equational proofs and first-order ones is studied in [Dershowitz and Plaisted, 1988] and a semantics based on *term logic* is studied in [Plaisted, 1987].

As for unconditional theories, we can now define the validity (is $s \leftrightarrow^* t$?) and satisfiability (does there exist σ such that $s\sigma \leftrightarrow^* t\sigma$?) problems, both of which are semi-decidable by brute-force enumeration methods. The interesting question is: How useful is the concept of conditional rewriting for solving these problems?

3.2 Some Translations of Conditional Equations

3.2.1 Conservative Extensions

In general, we can have more than one equation in the condition. But, we can always represent all the equations in the condition by a single equation using some new uninterpreted function symbols. We do this sometimes for notational convenience.

Let E be a set of conditional equations over terms $T(F, X)$. Let *true* be a new distinguished constant and *eq* and & be new binary operators (not in F). Let $F' = F \cup \{true, eq, \&\}$. We can convert E to a set of conditional equations E' over $T(F', X)$, where all conditions in E' are a single equation of the form $p = true$ as follows.

We replace a conditional equation

$$p_1 = q_1 \wedge \cdots \wedge p_n = q_n : s = t$$

by

$$(eq(p_1, q_1) \,\&\, \cdots \,\&\, eq(p_n, q_n)) = true : \; s = t$$

We also add the unconditional equations:

$$\begin{aligned} eq(x, x) &= true \\ x \,\&\, true &= x \end{aligned}$$

to E'.

With this translation, it is quite easy to show that for terms s and t in the original signature of E (that is, not having function symbols eq, & or $true$) $E \vdash s = t$ iff $E' \vdash s = t$. Such a translation is a conservative extension of the original theory.

3.2.2 Translating to Unconditional Equations

It may seem possible to simulate conditional equations by unconditional equations using a new binary operator if with the axiom $if(true, x) = x$. A conditional equation $c : s = t$ may then be translated to $s = if(c, t)$.

This translation is not sound, however, as illustrated in the following example. Consider E below:

$$\begin{array}{|lcccc|}\hline 0=1 & : & b & = & a \\ 0=1 & : & c & = & a \\ \hline\end{array}$$

The straightforward translation would yield an unconditional set of equations E':

$$\begin{array}{|rcl|}\hline eq(x, x) & = & true \\ if(true, x) & = & x \\ b & = & if(eq(0,1), a) \\ c & = & if(eq(0,1), a) \\ \hline\end{array} .$$

The following is a valid proof in E'

$$b \leftrightarrow if(eq(0,1), a) \leftrightarrow c$$

whereas $b \leftrightarrow^* c$ is not a valid consequence of the original theory E.

This translation mechanism can be modified slightly to make it sound and thus convert any set E of conditional equations to a set E' of unconditional equations. The change is to replace

equations of the form $c : s = t$ by $if(c, s) = if(c, t)$. With this method E' becomes:

$$\boxed{\begin{array}{rcl} eq(x, x) & = & true \\ if(true, x) & = & x \\ if(eq(0, 1), b) & = & if(eq(0, 1), a) \\ if(eq(0, 1), c) & = & if(eq(0, 1), a) \end{array}}$$

which is a (unconditional) conservative extension to E.

If we can, as above, translate any set of conditional equations to an equivalent unconditional conservative extension, then why study conditional rewriting at all? As we will see later in Chapter 5, this translation is not always good from the point of view of generating convergent rewrite systems equivalent to the equational theory. It often leads to generating infinite set of unconditional rules for theories that can be handled by finite conditional rewrite systems. But, this translation mechanism does come in very useful when some conditional equations cannot be handled by completion techniques. Selective translation of conditional equations to unconditional ones will be a very important part of completion methods that we will describe in Chapter 5.

3.3 Conditional Rewrite Systems

To make a conditional rule $c : l \rightarrow r$ from a conditional equation $c : s = t$ we have to do two things:

1. the equation in the conclusion must be oriented into a rule with the "bigger" term on the left;

2. a criterion must be chosen to determine whether a conditional rule applies (check if the equations in the condition "hold").

A conditional rule is used to rewrite terms by replacing an instance of the left-hand side with the corresponding instance of the right-hand side (but not in the opposite direction) provided the conditions hold. A set of conditional rules is called a conditional rewrite system. Depending on what criterion is used to check conditions, different rewrite relations are obtained for any given system R (see below). Once a criterion is chosen, we can define the one-step rewrite relation $\rightarrow$ and its reflexive-transitive closure $\rightarrow^*$ as follows: $u[l\sigma]_\pi \rightarrow u[r\sigma]_\pi$ if $c : l \rightarrow r$ is a rule, σ is a substitution, $u[l\sigma]_\pi$ is a term with subterm $l\sigma$ at position π $c\sigma$ satisfies the criterion.

There are a fair number of different ways of formulating conditional equations as rewrite rules:

Semi-Equational systems: Here we formulate rules as

$$p_1 = q_1 \ \wedge \ \cdots \ \wedge \ p_n = q_n \ : \ l \ \rightarrow \ r$$

where the conditions are still expressed as equations. To check if a condition holds we use the rules bidirectionally, as equations, and check if $p_i\sigma \leftrightarrow^* q_i\sigma$.

Join systems: Here we express rules as

$$p_1 \downarrow q_1 \ \wedge \ \cdots \ \wedge \ p_n \downarrow q_n \ : \ l \ \rightarrow \ r$$

The conditions are now checked in the rewrite system itself by checking if $p_i\sigma$ and $q_i\sigma$ are joinable in R itself by rewriting. Note the circularity in the definition of $\rightarrow$. The base case, of course, is when unconditional rules are used or the conditions unify syntactically. This definition is the one most often used; see [Kaplan, 1984; Jouannaud and Waldmann, 1986; Dershowitz et al., 1987].

Normal-Join systems: Here rules are written

$$p_1 \downarrow^! q_1 \ \wedge \ \cdots \ \wedge \ p_n \downarrow^! q_n \ : \ l \ \rightarrow \ r$$

This is similar to join systems except that $p_i\sigma$ and $q_i\sigma$ are not only joinable, but also have a common reduct that is irreducible. A sufficient condition for this is that the common reduct not contain any instance of a left-hand side. This is not a necessary condition, however, as an irreducible term may contain instances of the left-hand side of a rule, and for this instance the condition may not be feasible.

Normal systems: A special form of normal-join systems has all conditions of the form $p_i \xrightarrow{!} q_i$ (meaning that $p_i \rightarrow^* q_i$ and q_i is an irreducible ground term).

Meta-Conditional systems: Here we allow any (not necessarily recursively enumerable) predicate p in the conditions. For example, we may have conditions like $s \in S$ (for some term s and set S), $x \xrightarrow{!} x$ (i.e. x is already in normal form), or $l > r$ (for some ordering $>$). We write $p : l \rightarrow r$.

Most of the formulations above have been considered by different authors with slight variations. For example, Bergstra and Klop in [Bergstra and Klop, 1986] restrict their attention to systems which are *left-linear* (no left-hand side has more than one occurrence of any variable), have no "extra" variables in conditions and non-overlapping (no left-hand side unifies with a renamed non-variable subterm of another left-hand side or with a renamed proper subterm of itself). With these restrictions on left-hand sides, they refer to semi-equational systems as of *Type I*, join systems as of *Type II* and normal systems as of *Type* III_n. They also prove that, with these restrictions on left-hand sides, *Type I* and *Type* III_n systems are confluent. Meta-conditional systems with *membership* conditions were proposed in [Toyama, 1987].

For a given join system, we define the *rewrite* ($\rightarrow$) and *join* ($\downarrow$) relations on terms, as follows: Let $p \downarrow q : l \rightarrow r$ be a rule, s be a term, π be a position of a subterm in s, and σ be a substitution.

Then we say that the term $s[l\sigma]_\pi$, that is, the term s with an instance $l\sigma$ of the left-hand side l at position π, rewrites to the term $s[r\sigma]_\pi$ (s with $r\sigma$ in place of $l\sigma$) if $p\sigma$ and $q\sigma$ each rewrite in zero or more steps to the identical term; in that case, we say that σ is a *feasible* substitution for the rule.

As for unconditional systems, we write $s \to t$, if s rewrites to t in one step; $s \to^* t$, if s rewrites to t in zero or more steps, i.e. if t is *derivable* from s; $s \downarrow t$, if $s \to^* w$ and $t \to^* w$ for some term w; and $s \xrightarrow{!} t$, if $s \to^* t$, but no rewrite applies to t, i.e. the *normal form* t is derivable from s.

The following rules define $\leq$ on natural numbers:

		$0 \leq 0$	$\to$	tt
		$s(x) \leq 0$	$\to$	ff
		$s(x) \leq s(y)$	$\to$	$x \leq y$
$x \leq y \downarrow tt$	:	$x \leq s(y)$	$\to$	tt

For the above example, we have $0 \leq s(0) \xrightarrow{!} tt$ using the last rule, since the condition $0 \leq 0 \downarrow tt$ is achieved by the first rule.

The *depth* of a rewrite is the depth of recursive evaluations of conditions needed to determine that the matching substitution is feasible. Formally, the depth of an unconditional rewrite is 0; the depth of a rewrite using a conditional rule $p \downarrow q : l \to r$ and substitution σ is $depth(p\sigma \downarrow q\sigma) + 1$; the depth of a n-step derivation $s \to^* t$ is the maximum of the depths of each of the n steps; the depth of a "valley" $s \downarrow t$, joining at a term v, is the maximum of the depths of $s \to^* v$ and $t \to^* v$; and the depth of a zero-step derivation or valley is 0. We write $s \underset{k}{\to} t$ if $s \to t$ and the depth of the rewrite step is no more than k. Similarly $s \underset{k}{\xrightarrow{*}} t$ will mean that the maximum depth in that derivation is at most k. For example, $0 \leq 0 \underset{0}{\to} tt$, $0 \leq s(0) \underset{1}{\to} tt$, and $0 \leq s^n(0) \underset{m}{\to} tt$, for all $m \geq n$.

3.4 Termination of Conditional Rewriting

The notion of termination of conditional rewriting is similar to that of unconditional systems. Termination is sometimes confused with the *decidability* of rewriting. A rewriting system R is *terminating* (or $\rightarrow$ is noetherian) for a set of terms T if there are no infinite derivations $t_1 \rightarrow t_2 \rightarrow \cdots$ of terms in T. For the decidability of rewriting, we have to show that the recursive evaluation of terms in the condition terminates. We distinguish these two concepts below.

A sufficient condition for a rewrite system R to be terminating ($\rightarrow$ is noetherian) is that the unconditional version (dropping all conditions from rules) be so. We can use any reduction ordering to show this and do not need any restriction on the conditions. But, this is not strong enough to show termination of rules like $(x \geq 0) \downarrow tt : factorial(x) \rightarrow x * fact(x-1)$ where the unconditional version of the rule is not terminating. We really only need that $l\sigma \succ r\sigma$ in a well-founded ordering $\succ$ for those substitutions that are feasible (i.e. $c\sigma \downarrow tt$).

Rewriting is also decidable if for some well-founded ordering $\succ$ on terms and for each rule $c : l \rightarrow r$ we have $u[l\sigma] \succ c\sigma$ and

$$c\sigma \downarrow tt \quad implies \quad u[l\sigma] \succ u[r\sigma]$$

for all contexts $u[\cdot]$ and substitutions σ.

3.5 Strength of Rewrite Systems

Let E be a set of conditional equations. By $E \vdash s = t$, we mean that $s \leftrightarrow^* t$ is provable in E. Similarly, if R is a rewrite system (in any of the formulations), we use $R \vdash s \downarrow t$, to mean that s and t are joinable using the rules in R.

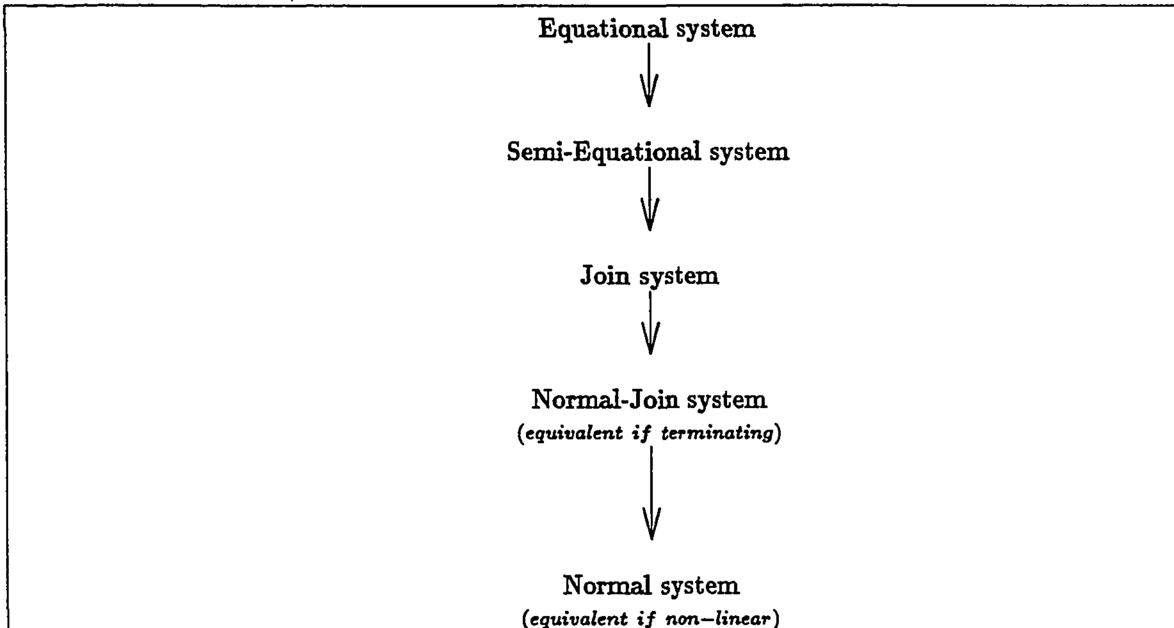

Figure 3.1: Logical Strength of Formulations

R and E have the same *logical strength* if $E \vdash s = t$ iff $R \vdash s \downarrow t$. Similarly, two rewrite systems R and R' have the same logical strength if $R \vdash s \downarrow t$ iff $R' \vdash s \downarrow t$. We say that R is *stronger* than R' if any two terms joinable using R' are joinable using R, but not the converse.

Figure 3.1 depicts the relative strength of the various formulations. In the figure, $A \rightarrow B$, means that A is stronger than B in general. That is, if we take a system of type B and just change the connective in conditions to convert to a system of type A (for example, $s \downarrow^! t$ to $s \downarrow t$ to convert an normal-join to a join system), then we have that what is provable in B is also provable in A. In particular, if B is convergent then so is A. The converse is, of course, not true in general. The relationships shown follow easily from the criterions used for checking if the condition "holds." For example, any condition that holds in the join formulation must also hold in the semi-equational formulation.

We now state and prove some of the equivalences and relationships between the various systems.

Proposition 3.1 *If a join system is terminating, then it is equivalent to the corresponding normal-join system (obtained by changing conditions of the form $s \downarrow t$ to $s \downarrow^! t$).*

Proposition 3.2 *A join system R can be converted to an equivalent normal join system R' by a conservative extension (using new function symbols) provided that we allow the normal system to be non-left-linear (have repeated variables in left-hand sides).*

Proposition 3.3 *Let R (with conditions of the form $s \downarrow t$) be a convergent (confluent and terminating) join system R' the corresponding semi-equational systems (change conditions to $s = t$) and E the underlying equational system (change conditions to $s = t$ and $l \rightarrow r$ to $l = r$). The following are equivalent:*

1. *$u = v$ is provable in E. That is, $E \vdash u = v$*
2. *u and v have a common reduct in R. That is, $R \vdash u \downarrow v$*
3. *u and v have a common reduct in R'. That is, $R' \vdash u \downarrow v$*

Proof: Proposition 1 is easy to see, for the termination property implies that if two terms are joinable, then they have a common reduct that is irreducible. The translation mechanism for Proposition 2 uses two new function symbols *eq* and *true*. We add a new rule $eq(x, x) \rightarrow true$ and change conditions to the form $eq(p_i, q_i) \downarrow true$. With this translation, it is easy to prove that for any two terms s and t not having the new function symbols *eq* and *true*, we have $R \vdash s \downarrow t$ iff $R' \vdash s \downarrow t$. The argument for Proposition 3 is by induction on the depth of a proof. The interesting case is when $u = v$ is provable in E and we wish to show $u \downarrow v$ in R. By induction on the depth we first show that the subproofs in E can be replaced by rewrite proofs and then using the confluence of

R we can show that $u \downarrow v$. □

Under the assumption of convergence, the weaker formulations of join systems are also equivalent to the corresponding join system and, hence, to the underlying equational system.

3.6 Decreasing Systems

In this section, we will use the join system formulation of conditional rules. By the *reduction ordering* $\succ_R$ of a rewrite system R, we mean the irreflexive-transitive closure $\rightarrow^+$ of the rewrite relation ($\rightarrow$). That is, $t_1 \succ_R t_2$ if $t_1 \rightarrow^+ t_2$. The reduction ordering is *monotonic*. That is, if $t \succ_R s$ then $u[t] \succ_R u[s]$ for any context $u[\cdot]$. By the *proper subterm ordering* $\succ_s$ we mean the well-founded ordering $u[t] \succ_s t$ for any term t and non-empty context $u[\cdot]$.

A conditional rewrite system is *decreasing* if there exists a well-founded extension $\succ$ of the proper subterm ordering $\succ_s$, such that $\succ$ contains $\succ_R$ and $l\sigma \succ p_1\sigma, \ldots, q_n\sigma$ for each rule

$$p_1 \downarrow q_1 \ \wedge \ \cdots \ \wedge \ p_n \downarrow q_n \ : \ l \ \rightarrow \ r$$

($n \geq 0$) and for any substitution σ.

Note that the second condition restricts all variables in the condition to also appear on the left-hand side. In general, a decreasing ordering need not be monotonic.

Proposition 3.4 *If a rewrite system is decreasing, it has the following properties:*

1. *The system is terminating.*

2. *The basic notions are decidable. That is, for any terms s, t*

   (a) *one-step reduction ("does $s \rightarrow t$?")*

(b) finite reduction ("does $s \rightarrow^* t$?")

(c) joinability ("does $s \downarrow t$?")

(d) normal form or reducibility ("is s irreducible?")

are all decidable.

Proof: That the system is terminating is obvious from the well-foundedness of $\succ$. The decidability of basic notions is proved by transfinite induction on $\succ$, as follows. We first consider the following property: "Given a term t we can find the set of normal forms of t." If t has no instance of a left-hand side of any rule as a subterm, then t is irreducible and it is its only normal form. Otherwise, let $t = u[l\sigma]$ for some rule

$$p_1 \downarrow q_1 \ \wedge \ \cdots \ \wedge \ p_n \downarrow q_n \ : \ l \ \rightarrow \ r$$

By our two conditions on decreasingness we have that $t = u[l\sigma] \succ l\sigma$ and $l\sigma \succ p_i\sigma, q_i\sigma$. By induction, since $t \succ p_i\sigma, q_i\sigma$, we can compute the set of normal forms of $p_i\sigma, q_i\sigma$ for each i and check if the rule applies. If it does, then $t \rightarrow u[r\sigma]$. Similarly (using each matching rule) we can compute all the terms, say $s_1, \ldots, s_n$, that t rewrites to in one-step. By induction hypothesis, one can enumerate the normal forms for each s_i. The union of these is the set of normal forms for t.

Other basic properties can be shown, likewise, decidable. □

Not only is decreasingness a sufficient condition for making many basic notions of rewriting decidable, it also captures exactly the class of rules for which the recursive evaluation of conditions is finite [Dershowitz and Okada, 1988]. We use the following strategy to simplify a term s to *normal* form. If

$$p_1 \downarrow q_1 \ \wedge \ \cdots \ \wedge \ p_n \downarrow q_n \ : \ l \ \rightarrow \ r$$

is a rule such that $l\sigma$ is a subterm of s, then we recursively compute the normal forms of the $p_i\sigma$ (and $q_i\sigma$) to see if the rule applies. The class of rules, for which this strategy terminates, must be decreasing.

More precisely, let $\rhd$ be a relation on terms such that $s \rhd p$ if p is a term that we may have to recursively normalize when trying to find the normal form of s (that is, all the instances of conditions $p_i\sigma$s and $q_i\sigma$s that we evaluate whenever a left-hand side matches a subterm of s). Let R be a conditional rewrite system and $\rightarrow$ its rewrite relation.

Lemma 3.1 $(\rightarrow \cup \rhd)^+$ *is well-founded if, and only if, R is decreasing.*

Proof: The "if" direction is a direct consequence of the definition of decreasingness. The decreasing ordering $\succ$ is, by definition, $(\rightarrow \cup \rhd \cup \succ_s)^+$, where $\succ_s$ is the proper subterm ordering. If $\succ$ is well-founded, then so is $(\rightarrow \cup \rhd)^+$.

To show the other direction, we first note that

1. If $u \succ_s v \rightarrow w$, then $u \rightarrow v' \succ_s w$. That is, if a subterm of u rewrites to w, then u must rewrite to a superterm of w.

2. If $u \succ_s v \rhd w$, then $u \rhd w$. That is, if w is a term we evaluate when normalizing a subterm of u, then w may also be evaluated when normalizing u.

3. If $u \succ_s v \succ_s w$, then $u \succ_s w$. That is, the proper subterm relation is transitive.

Thus, $(\rightarrow \cup \rhd \cup \succ_s)^* \subset (\rightarrow \cup \rhd)^* \succ_s{}^*$. So, if $(\rightarrow \cup \rhd)^+$ is well-founded, that is, the recursive evaluation of the conditions is finite (or R is in the class of terminating programs), then R must be decreasing.

The following are sufficient conditions for decreasingness:

Simplifying systems: [Kaplan, 1984; Kaplan, 1987] A conditional rewrite system R is *simplifying* if there exists a simplification ordering $\succ$ (in the sense of [Dershowitz, 1987]) such that $l\sigma \succ r\sigma, p_1\sigma..., q_n\sigma$, for each rule

$$p_1 \downarrow q_1 \ \wedge \ \cdots \ \wedge \ p_n \downarrow q_n \ : \ l \ \rightarrow \ r$$

$(n \geq 0)$.

Reductive systems: [Jouannaud and Waldmann, 1986] A conditional rewrite system is *reductive* if there is a well-founded monotonic ordering $\succ$ such that $\succ$ contains the reduction ordering $\succ_R$ and $l\sigma \succ p_1\sigma, ..., q_n\sigma$ for each rule

$$p_1 \downarrow q_1 \ \wedge \ \cdots \ \wedge \ p_n \downarrow q_n \ : \ l \ \rightarrow \ r$$

$(n \geq 0)$.

Both simplifying systems and reductive systems are special cases of decreasing ones. To see this for simplifying systems, note that simplification orderings contains the subterm ordering, by definition. For reductive systems, note that no monotonic well-founded ordering can have $s \succ t$ for a proper subterm s of t. So we can extend the monotonic ordering with the subterm property and get a well-founded ordering as in [Jouannaud and Waldmann, 1986].

The following is an example of a system that is decreasing, but neither simplifying nor general reductive:

	b	$\rightarrow$	c
	$f(b)$	$\rightarrow$	$f(a)$
$b \downarrow c$:	a	$\rightarrow$	c

This is not reductive, because there can be no monotonic extension of the reduction ordering (which has $f(b) \succ_R f(a)$) that can have $a \succ b$.

4 CONFLUENCE OF CONDITIONAL REWRITE SYSTEMS

In this chapter, we study the confluence of conditional rewrite systems. We will mostly restrict our attention to noetherian rewrite systems and use the join formulation of rewriting. For the class of decreasing systems, we find that results on confluence correspond directly to those for unconditional rewriting. In particular, it is sufficient to check if all the critical pairs between rules are joinable. But, when we consider non-decreasing systems, we run into difficulties. We give a counter-example to show that the joinability of critical pairs is not sufficient to guarantee confluence even when the system is terminating. We then examine some restrictions like "shallow joinability" of critical pairs and "overlay" systems for which we give positive results on confluence.

4.1 Local Confluence and Critical Pairs

A rewrite relation (conditional or unconditional) is *confluent* if whenever two terms, s and t, are derivable from a term u, then a term v is derivable from both s and t. A relation $\rightarrow$ is *locally confluent*, if $s \rightarrow^* v$ and $t \rightarrow^* v$ for some v whenever $u \rightarrow s$ and $u \rightarrow t$ (in one step).

For terminating relations, we have the following well known lemma–

Diamond Lemma [Newman, 1942] A terminating relation is confluent iff it is locally confluent.

How can we check the local confluence of a (terminating) conditional rewrite relation? We consider below the three cases that arise when we apply two different rewrites to a term t producing terms t_1 and t_2. Such a situation $t_1 \leftarrow t \rightarrow t_2$ is called a *peak*.

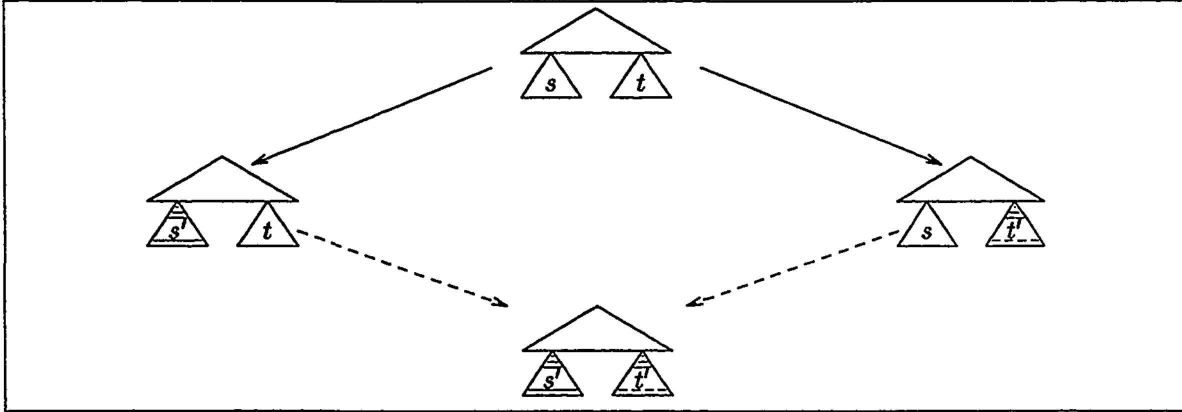

Figure 4.1: Disjoint Peak

4.1.1 Disjoint Peaks

If a term $u[s,t]$ contains subterms s and t at independent positions (neither term occurs as a subterm of the other), and $s \to s'$ and $t \to t'$, then the resultant terms, $u[s',t]$ and $u[s,t']$, rewrite in one step to the same term, $u[s',t']$ using the same rules and substitutions. The situation $u[s',t] \leftarrow u[s,t] \to u[s,t']$ is called a *disjoint peak*. See Figure 4.1.

4.1.2 Variable Peaks

Let τ be a feasible substitution for a rule $p' \downarrow q' : g \to d$ and let σ be a feasible substitution for a rule $p \downarrow q : l \to r$ under which some variable x in l is mapped to a term $c[g\tau]$, containing a rewritable instance of g. Then, the term $l\sigma$ can be rewritten in two different ways, to $r\sigma$ and $l\sigma[d\tau]$, as depicted in Figure 4.2. We refer to this as a *variable peak*.

If l is non-linear in x, then each of the remaining occurrences of $c[g\tau]$ in $l\sigma$ may be rewritten until a term $l\sigma'$ is obtained, where σ' is same as σ except that x is mapped to $c[d\tau]$. Similarly, if r is non-linear in x then we need additional rewrites to get $r\sigma \to^* r\sigma'$. When dealing with unconditional rewriting systems, variable peaks are always joinable, since $l\sigma' \to r\sigma'$. But for conditional systems,

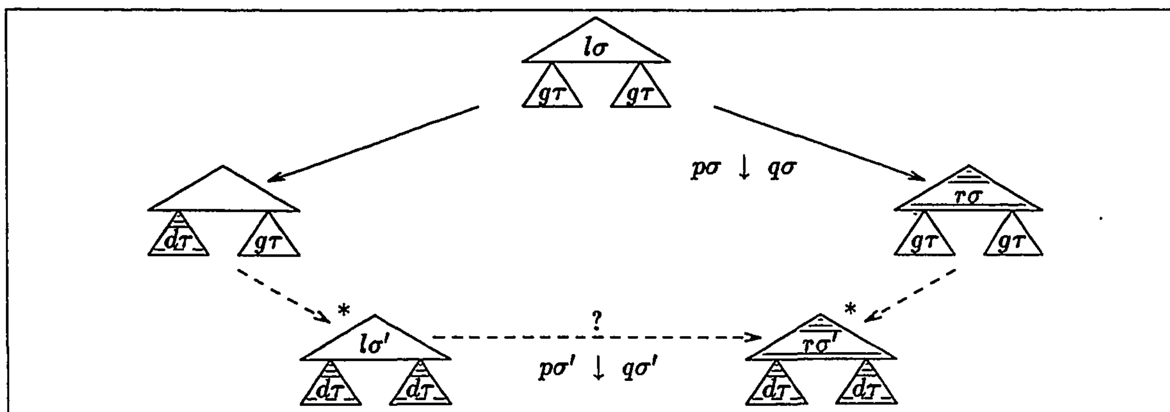

Figure 4.2: Variable Overlap

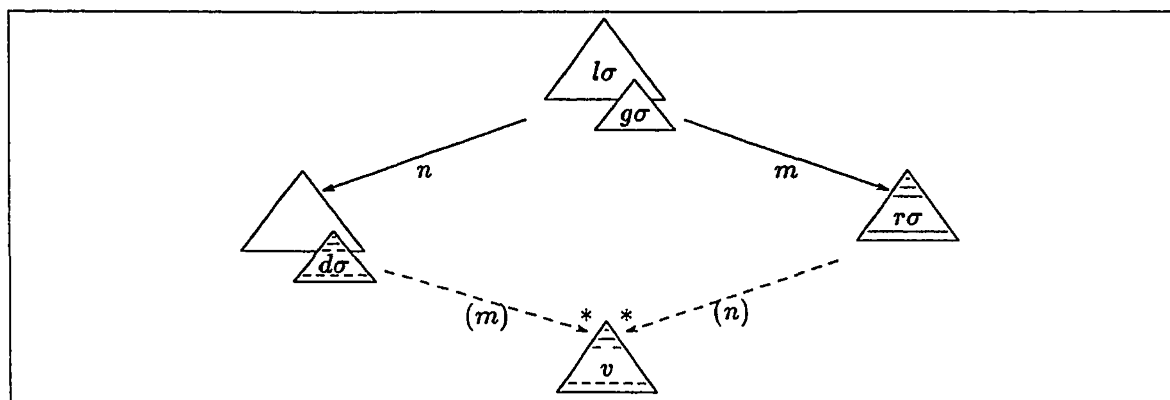

Figure 4.3: (Shallow) Joinable Critical Peak

σ' must also be feasible, i.e. $p\sigma' \downarrow q\sigma'$ must hold for the rule to apply. This is not always the case, even if critical pairs are joinable, as the counter-examples of the next section will demonstrate.

4.1.3 Critical Peaks and Conditional Critical Pairs

If the left-hand side g of a rule $p' \downarrow q' : g \to d$ unifies, via most general substitution σ, with a non-variable subterm s at position π in a left-hand side l of a rule $p \downarrow q : l \to r$, then the conditional equation

$$p\sigma = q\sigma \ \wedge \ p'\sigma = q'\sigma : \ l\sigma[d\sigma]_\pi = r\sigma$$

is called a (conditional) *critical pair* of the two rules, where $l\sigma[d\sigma]_\pi$ is obtained by replacing s in l by d and applying σ.

The situation

$$l\sigma[d\sigma]_\pi \longleftarrow l\sigma \longrightarrow r\sigma$$

is called a *critical overlap*, and, for any context u and substitution τ less general than σ,

$$u[l\tau[d\tau]_\pi] \longleftarrow u[l\tau] \longrightarrow u[r\tau]$$

is called a *critical peak*. See Figure 4.3.

For example, the following rules:

$member(x,y) \downarrow ff$	:	$delete(x,y)$	$\rightarrow$	y
$different(x,z) \downarrow tt$	:	$delete(x, z \cdot y)$	$\rightarrow$	$y \cdot delete(x,y)$

have the critical pair–

$$different(x,z) = tt \ \wedge \ member(x, z \cdot y) = ff \ : \ z \cdot delete(x,y) = z \cdot y$$

A critical pair $c = d : s = t$ is *feasible*, and the corresponding overlap is *feasible*, if there is a substitution σ for which $c\sigma \downarrow d\sigma$. A *trivial* critical pair is one for which s is identical to t. A system is *non-overlapping* (unambiguous) if it has no feasible, non-trivial critical pairs.

With the usual definition of *even* and *odd* the critical pair–

$$odd(x) = tt \wedge even(x) = tt : s(0) = 0$$

between the rules:

$$odd(x) \downarrow tt \ : \ f(x) \ \rightarrow \ 0$$
$$even(x) \downarrow tt \ : \ f(x) \ \rightarrow \ s(0)$$

is infeasible.

		b	$\rightarrow$	$f(b)$
$x \downarrow f(x)$	:	$f(x)$	$\rightarrow$	a

Table 4.1: Example A (Left-linear and non-overlapping, but not normal or terminating)

A critical pair $c = d : s = t$ is *joinable* if $s\sigma \downarrow t\sigma$ for any substitution σ such that $c\sigma \downarrow d\sigma$. Infeasible critical pairs are vacuously joinable and trivial ones are trivially joinable.

A critical pair $c = d : s = t$, obtained from a critical overlap $s \leftarrow u \rightarrow t$ is *shallow-joinable*, if, for each substitution σ such that $c\sigma \downarrow d\sigma$, there exists a term v such that $s\sigma \xrightarrow[n]{*} v$ and $t\sigma \xrightarrow[m]{*} v$, where m is the depth of $u \rightarrow s$ and n is the depth of $u \rightarrow t$. A conditional rewrite system is *shallow-joinable* if each of its critical pairs is.

In other words, every critical pair of a shallow-joinable system joins with the corresponding depths less or equal to those of the critical overlap. (See Figure 4.3). In particular, critical pairs between unconditional rules must be unconditionally joinable.

4.2 Counter-Examples

In this section, we present non-confluent systems that are counter-examples to attempts at extending theorems for unconditional systems to the analogous conditional case.

Unconditional systems are locally confluent, if all their critical pairs are joinable. On the other hand, Example A [Bergstra and Klop, 1986] shows that non-normal, non-terminating conditional systems need not be locally confluent, even if they are left-linear and non-overlapping. In that example, the term b has many normal forms, including a and $f(a)$, despite the lack of critical pairs.

		a	$\to$	b
		c	$\to$	$k(f(a))$
		c	$\to$	$k(g(b))$
		$h(x)$	$\to$	$k(x)$
		$h(f(a))$	$\to$	c
$h(f(x)) \downarrow k(g(b))$	:	$f(x)$	$\to$	$g(x)$

Table 4.2: Example B (Terminating, left-linear and normal, but not shallow-joinable)

		$p(b)$	$\to$	$k(f(a))$
		$q(b)$	$\to$	$k(g(b))$
$b \downarrow b$	:	a	$\to$	b
$x \downarrow a$	:	$p(x)$	$\to$	$q(x)$
$b \downarrow b$	:	$h(x)$	$\to$	$k(x)$
$b \downarrow b$	:	$h(f(a))$	$\to$	$p(a)$
$h(f(x)) \downarrow k(g(b))$	:	$f(x)$	$\to$	$g(x)$

Table 4.3: Example C (Terminating, left-linear and shallow-joinable, but not normal)

		$eq(x, x)$	$\to$	$true$
		$p(b)$	$\to$	$k(f(a))$
		$q(b)$	$\to$	$k(g(b))$
$b \downarrow b$	:	a	$\to$	b
$b \downarrow b$	:	$h(x)$	$\to$	$k(x)$
$b \downarrow b$	:	$h(f(a))$	$\to$	$p(a)$
$eq(x, a) \downarrow true$	:	$p(x)$	$\to$	$q(x)$
$h(f(x)) \downarrow k(g(b))$	:	$f(x)$	$\to$	$g(x)$

Table 4.4: Example D (Terminating, normal and shallow-joinable, but not left-linear)

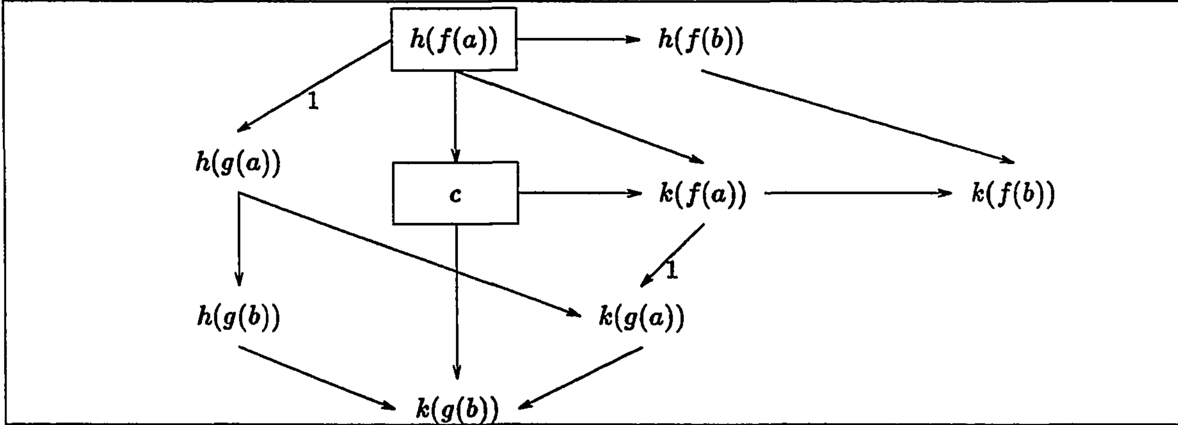

Figure 4.4: Critical pairs of Example B

As will be shown later (in Section 4.4), terminating conditional systems with no critical pairs are locally confluent. Unfortunately:

Proposition 4.1 *There exists a terminating, non-locally-confluent, conditional rewrite system all of whose critical pairs are joinable.*

This is demonstrated by Example B. All four critical pairs:

1. $k(f(a)) = k(g(b))$

2. $h(f(b)) = c$

3. $k(f(a)) = c$

4. $h(f(a)) = k(g(b)) : \ h(g(a)) = c$

are joinable. But, the term $f(a)$ has two normal forms, $f(b)$ and $g(b)$. This is shown in Figure 4.4, where critical overlaps are boxed. Note that the unconditional critical pair, obtained by rewriting c in two ways, is joinable only using the conditional rule, i.e. it is not shallow-joinable.

With slight modifications, one obtains counter-examples C and D, showing that no combination of two of the following three factors suffices for confluence: left-linear, normal, and shallow-joinable.

From these examples, it is clear that we need relatively strong restrictions on rewrite systems to guarantee confluence. In Section 4.4, we will show that combining all three factors does, in fact, yield confluence for terminating systems.

4.3 Confluence of Decreasing Systems

In this section, we will show that for decreasing systems the "critical pair lemma" holds just like for unconditional systems. This generalizes results in [Kaplan, 1987] for simplifying systems and [Jouannaud and Waldmann, 1986] for reductive systems. We also use a general proof normalization technique to show this result by showing the equivalence of the semi-equational and join formulations.

Let us recall that a system is *decreasing* if there exists a well-founded ordering $\succ$, containing the proper subterm ordering $\succ_s$, such that $s \succ t$ whenever $s \rightarrow t$ and, for each rule $p \downarrow q : l \rightarrow r$ and substitution σ, $l\sigma \succ p\sigma$ and $l\sigma \succ q\sigma$. This is stronger than the terminating requirement. Examples B-D above are all terminating, but are not decreasing, since the left-hand side $f(x)$ is a proper subterm of the term $h(f(x))$ in the condition.

Also, the examples all use the join formulation of the conditions. All critical pairs are joinable, yet the term $f(a)$ has two normal forms $g(b)$ and $f(b)$ in Example B. We can use the last rule to rewrite $f(a)$ to $g(a)$ because the condition for this substitution $\{x \mapsto a\}$ has a rewrite proof $h(f(a)) \rightarrow c \rightarrow k(g(b))$. But to show $f(b) \rightarrow g(b)$ using this same rule we have to check the condition $h(f(b)) \downarrow k(g(b))$ which leads to a cycle. Note that if we converted this to a semi-equational system, we *would* have that $h(f(b)) \leftrightarrow^* k(g(b))$; the last rule can be applied and the system is confluent.

First, we observe that for the semi-equational formulation of conditional rewriting the critical pair lemma holds.

Lemma 4.1 *For any semi-equational conditional rewrite system, if every critical pair is joinable, then the system is confluent.*

Proof: The variable peaks (Figure 4.2) present no problem in the semi-equational formulation of conditional rewriting. This is because if $p\sigma \leftrightarrow^* q\sigma$ and $p\sigma \leftrightarrow^* p\sigma'$ and $q\sigma \leftrightarrow^* q\sigma'$ then we do have a proof $p\sigma' \leftrightarrow^* q\sigma'$ in the underlying theory and the variable peak can be made joinable by rewriting.

□

We saw in the previous chapter that, while the confluence of a join system implies the confluence of the corresponding semi-equational system (without any other restriction), the converse is not true. We now show that, if we restrict our attention to decreasing systems, the converse does hold. That is, under the assumption of decreasingness, the two formulations—semi-equational systems and join systems—are equivalent with respect to confluence.

Theorem 4.1 *If a decreasing semi-equational system (conditions of the form $s = t$) is confluent, then the corresponding join system (with conditions changed to $s \downarrow t$) is also confluent.*

It is convenient to introduce the following notations. By a *direct proof* of $s = t$, we mean a rewrite proof of the form $s \downarrow t$. That is, s and t are joinable. By a *completely direct* proof of $s = t$, we mean a direct proof of $s = t$ (i.e, of $s \downarrow t$) in which every subproof of the conditions (during application of conditional rules) is also completely direct. For instance, if a substitution instance of the form–

$$s_1\sigma = t_1\sigma \ \wedge \cdots \wedge \ s_n\sigma = t_n\sigma : \ l\sigma \rightarrow r\sigma$$

of a conditional rule is used in the proof with subproofs of $s_i\sigma = t_i\sigma$, then each of these subproofs is also completely direct. If for a given proof P of $s = t$ there is a completely direct proof P' (of $s \downarrow t$), then we say that the proof P is completely normalizable.

Lemma 4.2 *(Complete Normalizability Lemma) For any confluent and decreasing semi-equational system, every proof is completely normalizable.*

Proof: This is proved using transfinite induction on the well-founded decreasing ordering. It is easily seen that every proof in a decreasing system can be made direct if the system is confluent. By using the properties of decreasingness, we can show that every top-level subproof is smaller in the decreasing ordering and, hence, can be made completely direct by the induction hypothesis. This lemma follows. □

Decreasing systems also satisfy the critical pair lemma.

Theorem 4.2 *For any decreasing system, if every critical pair is joinable, then the system is confluent, hence canonical.*

This theorem is a direct consequence of the Complete Normalizability Lemma and is essentially the same as the result in [Kaplan, 1987]. Thus, if a decreasing system is confluent in the semi-equational formulation, then it is confluent as a join system.

Also, attempts to weaken the definition of decreasing systems do not work. The counter-examples B-D satisfy the conditions for decreasing systems except the subterm property. The reduction ordering of those systems is embeddable into the well-founded ordering $<_\infty$ (which, however, does not have the subterm property) of Takeuti's system $O(2,1)$ of ordinal diagrams. Also, $<_\infty$ satisfies the additional condition for decreasingness (each term in the condition— d and

$h(f(x))$—is smaller than the left-hand side—$f(x)$—of that rule). So it is clear that well-foundedness alone is insufficient. For details see [Okada 1987; Dershowitz and Okada 1988]. In the next chapter on completion methods, we will see that decreasing rules are easy to handle, while special techniques are necessary for non-decreasing equations and rules.

Decreasing systems cannot handle rules containing variables in the condition that do not also appear on the left-hand side. But in the programming context, at least, one would certainly like to allow rules such as:

$$(x \leq y, y \leq z) \downarrow (tt, tt) : \; x \leq z \rightarrow tt$$

where y is an "extra" variable, or:

$$fib(x) \downarrow < y, z >: \; fib(s(x)) \; \rightarrow < y + z, y >$$

where the right-hand side also has an occurrence of the new variables, y and z.

As we saw in the previous chapter, operationally, rewriting is more difficult now, since new variables in the conditions must be solved for. Thus, to rewrite an instance $l\sigma$ of a left-hand side, an interpreter must first find a *satisfying* substitution τ for the new variables in the condition $p \downarrow q$ such that $p\sigma\tau$ joins $q\sigma\tau$, and then replace $l\sigma$ by $r\sigma\tau$. One way to enumerate solutions (for decreasing and confluent systems) is via (conditional) *narrowing* (see Chapter 5). Unfortunately, it is undecidable, in general, whether such a substitution exists.

Note, also, that with new variables on the right, a rule may non-trivially overlay itself. For example, a rewrites to $f(b)$ and $f(c)$ with the system

$$\begin{array}{rcrcl} & & p(b) & \rightarrow & tt \\ & & p(c) & \rightarrow & tt \\ p(x) \downarrow tt & : & a & \rightarrow & f(x) \end{array}$$

In general, a rule with new right-hand side variables can rewrite (in one step) to an infinite number of different terms. The next two sections give positive results for confluence for systems that include some of the cases above.

4.4 Confluence of Left-Linear, Normal Systems

In this section, we consider restrictions that ensure that a normal, left-linear system is confluent. Such systems arise naturally in pattern-directed functional languages, when the different cases are constructor-based and mutually exclusive.

Bergstra and Klop have shown the following for conditional systems that are not necessarily terminating:

Theorem 4.3 (Bergstra and Klop, 1986) *A left-linear, normal conditional rewrite system is confluent, if it is non-overlapping.*

(Though we have weakened their definition of non-overlapping to allow infeasible overlaps, the result still holds.) This is analogous to the standard result that left-linear unconditional systems with no critical pairs are confluent [Huet, 1980].

We give a similar result, for overlapping systems, in which critical pairs are shallow-joinable. For this, we require that the system be terminating. From the counter-examples of the previous section, one can see that this is optimal.

Theorem 4.4 *A terminating, left-linear, normal conditional rewrite system is confluent, if all its critical pairs are shallow joinable.*

This theorem is a corollary of the following:

Lemma 4.3 *Let R be a terminating conditional rewrite system that is left-linear, normal and shallow-joinable. Then, if $u \underset{m}{\rightarrow^*} s$ and $u \underset{n}{\rightarrow^*} t$, there exists a term v such that $s \underset{n}{\rightarrow^*} v$ and $t \underset{m}{\rightarrow^*} v$.*

Proof: The proof is by transfinite induction on the pair $\langle m+n\,,\, u\rangle$ with respect to the (lexicographic combination of the) natural ordering of natural numbers and the terminating relation $\rightarrow$ on terms.

Let $u \underset{m}{\rightarrow} s' \underset{m}{\rightarrow^*} s$ and $u \underset{n}{\rightarrow} t' \underset{n}{\rightarrow^*} t$. That is, u is first rewritten at position π to s' using rule $p \overset{!}{\rightarrow} N : l \rightarrow r$ with depth no greater than m, and at position π' to t' using $q \overset{!}{\rightarrow} M : g \rightarrow d$ with maximum depth n (M and N are normal forms). We show that s' and t' are joinable with appropriate depths at some term w. As in the Diamond Lemma (Section 4.1), two inductions (at the peaks, s' and t') show that s and t are also joinable with suitable depths.

If the peak at u is *disjoint*, then s' and t' join at a term w (which is u after both rewrites). That is, $t' \underset{m}{\rightarrow} w$ (by rewriting at π') and $s' \underset{n}{\rightarrow} w$ (by rewriting at π).

If the peak is *critical*, then, by the shallow-joinable assumption, there is a term w such that $t' \underset{m}{\rightarrow^*} w$ and $s' \underset{n}{\rightarrow^*} w$.

This leaves only the *variable peak* case. Without loss of generality, let π be above π'. Thus, some variable x in l matches a subterm $c[g\tau]$ which rewrites to $c[d\tau]$. Let σ' be the same as the substitution σ used in rewriting $u \rightarrow s'$, except that x is mapped to $c[d\tau]$. As seen in Figure 4.5, because R is left-linear, the subterm of t' at π is actually $l\sigma'$. Furthermore, $s' \underset{n}{\rightarrow^*} u[r\sigma']_\pi = w$, by rewriting all (zero or more) occurrences of $g\tau$ in s' to $d\tau$. It remains to show that $l\sigma' \underset{m}{\rightarrow} r\sigma'$ is feasible. Since the system is normal, we have $p\sigma' \underset{m-1}{\overset{!}{\rightarrow}} N$, as is required, by induction from the shallower peak $p\sigma' \,{}^*\!\!\leftarrow_n\, p\sigma \underset{m-1}{\overset{!}{\rightarrow}} N$. □

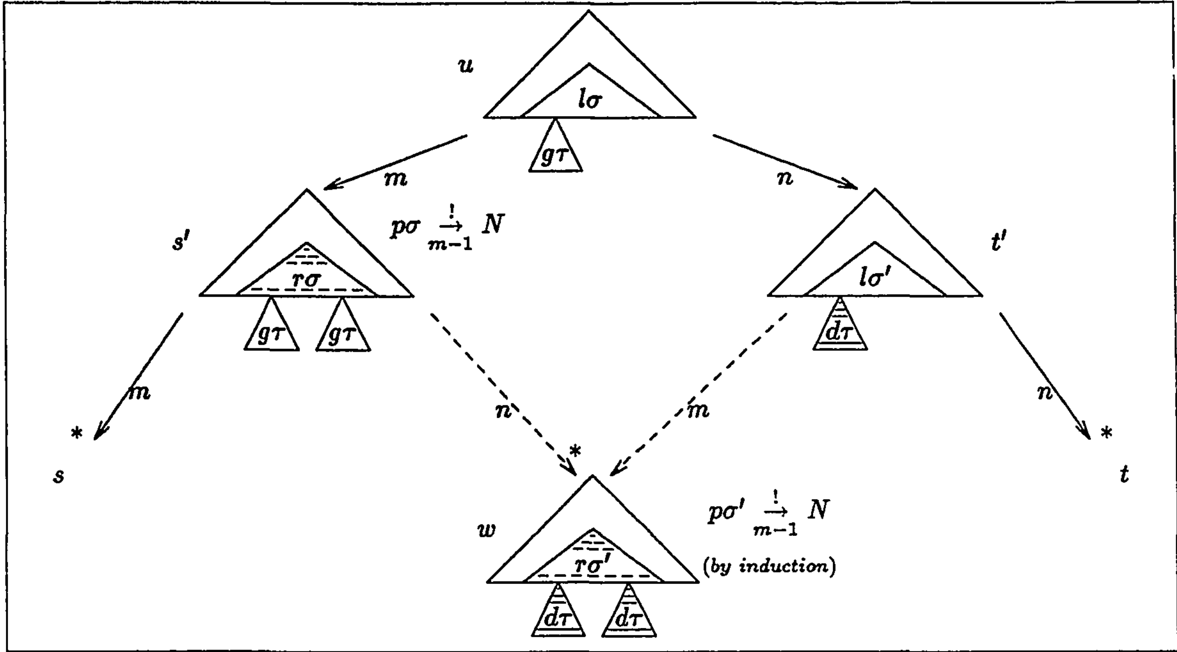

Figure 4.5: Normal, left-linear, variable overlap case

4.5 Confluence of Overlay Systems

In this section, we make no restrictions on the depth of the joinability of the critical pairs. We also do not insist on left-linearity and allow terms in the condition that are "bigger" than the left-hand side. Under certain circumstances, we are able to prove that such systems are confluent as long as all their critical pairs are joinable. This is close in spirit to the result for unconditional systems.

The only restriction we have is that we insist that overlaps between left-hand sides do not occur at proper subterms of the overlapped left-hand side. In particular, non-deterministic pattern-directed languages, with no nested defined function symbols in the patterns, meet this requirement.

A critical pair is an *overlay* if it is obtained from two left-hand sides that unify at their roots. In our original example, the critical pair $s(x) \leq y = tt : x \leq y = tt$ between the rules $s(x) \leq s(y) \rightarrow x \leq y$ and $x \leq y \downarrow tt : x \leq s(y) \rightarrow tt$ is an overlay.

Theorem 4.5 *A terminating conditional rewrite system is confluent, if all its critical pairs are joinable overlays.*

A particular consequence of this theorem is that any conditional system that is terminating and non-overlapping is confluent (as is the case for unconditional systems).

This theorem is a consequence of the following:

Lemma 4.4 *Let $u[s]_\Pi$, where Π is a set of positions, denote the term u with each subterm at a position in Π replaced by s. Let R be a terminating system in which all critical pairs are joinable overlays. If a term v is derivable from $u[s]_\Pi$ and t from s, then v and $u[t]_\Pi$ are joinable. (That is, if $u[s]_\Pi \rightarrow^* v$ and $s \rightarrow^* t$ then, $v \downarrow u[t]_\Pi$.)*

Proof: We show that if $u[s]_\Pi \underset{n}{\rightarrow}^* v$ and $s \rightarrow^* t$, then $u[t]_\Pi \downarrow v$, by induction on the triple $\langle s, n, u[s] \rangle$, where the first component is compared using the union of the terminating rewrite relation $\rightarrow$ and the proper subterm relation $\succ_s$, the second as a natural number, and the third by the rewrite relation.

If $u = v$ or $s = t$, then we are done. Otherwise, let $s \rightarrow s' \rightarrow^* t$. If we can show that $u[s']_\Pi \downarrow v$, then by induction it will follow that $u[t]_\Pi \downarrow v$, since s' is less (vis-a-vis $\rightarrow$) than s.

If the first rewrite $s \rightarrow s'$ occurs at a proper subterm $g\tau$ of s, then by induction on the first component, we have $u[s']_\Pi \downarrow v$. See Figure 4.6.

Otherwise, we may suppose that s is $g\sigma$ and s' is $d\sigma$, for some rule $p' \downarrow q' : g \rightarrow d$. Let $u \underset{n}{\rightarrow} u' \underset{n}{\rightarrow}^* v$, with the first step via rule $p \downarrow q : l \rightarrow r$ at position π. If this is a disjoint peak (i.e.

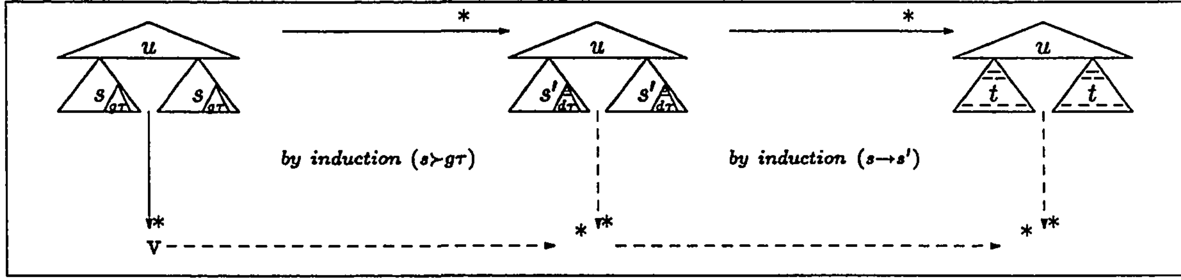

Figure 4.6: Proper subterm case

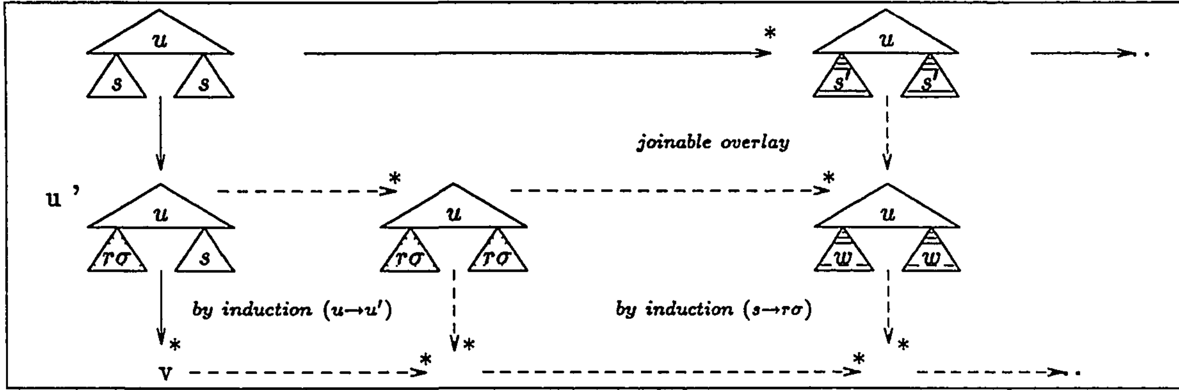

Figure 4.7: Overlay case

if π is not above or below any position in Π), then $u[s']_\Pi$ rewrites (at position π) to $u'[s']_\Pi$. Since u' is smaller than u, we have that $u'[s']_\Pi \downarrow v$. Thus, $u[s']_\Pi \rightarrow u'[s']_\Pi \downarrow v$.

If $u' \leftarrow u[s]_\Pi \rightarrow^* u[s']_\Pi$ is a critical peak, then it must be an overlay and $s = g\tau = l\sigma$. Critical pairs are joinable, so let $s' = d\tau \rightarrow^* w$ and $r\sigma \rightarrow^* w$, for some w. Then we have that $u' = u[r\sigma]_\pi \rightarrow^* u[r\sigma]_\Pi$. By induction on the last component, we have that $u[r\sigma]_\Pi \downarrow v$; by induction on the first, we have $u[w]_\Pi \downarrow v$. Thus, $u[s']_\Pi \rightarrow^* u[w]_\Pi \downarrow v$. This case is depicted in Figure 4.7.

The remaining case is that of a *variable overlap*, either above or in some s. Let π be above s; that is, some variable x in l matches a term $c[s]_{\Pi'}$ containing any number of occurrences of s. Let σ' be the same as the substitution σ used to rewrite $u \rightarrow u'$ except that x is mapped instead to

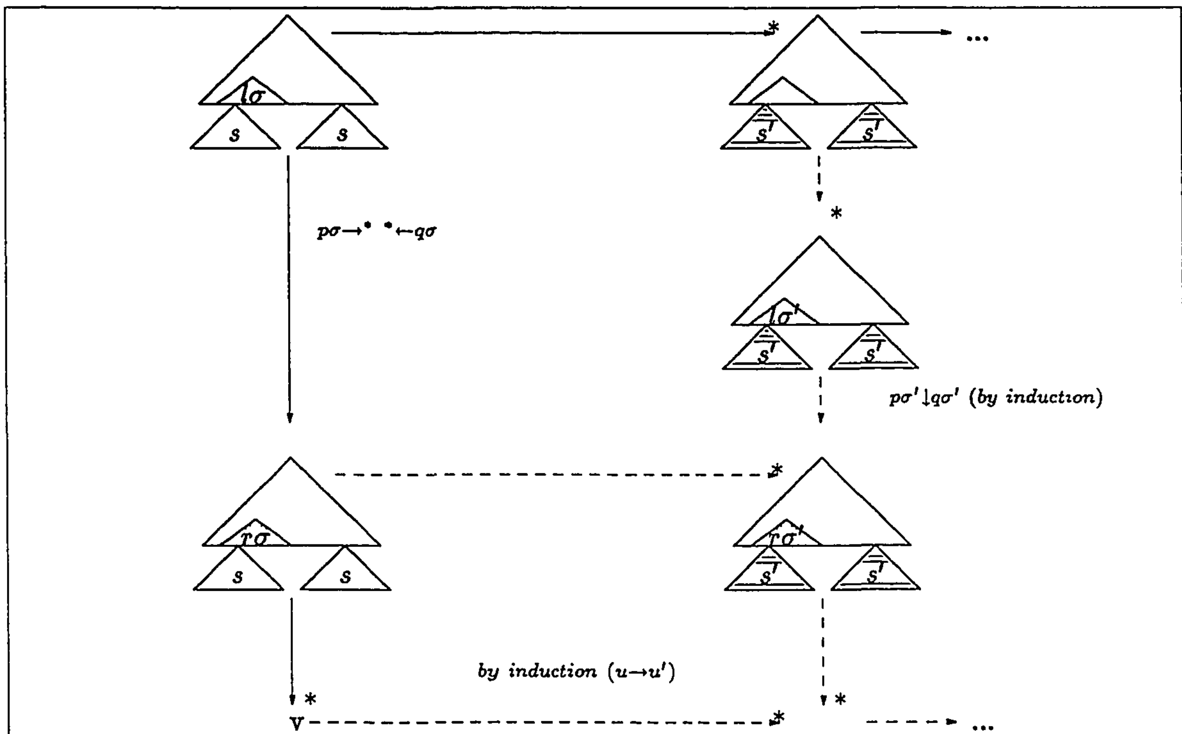

Figure 4.8: Variable overlap above

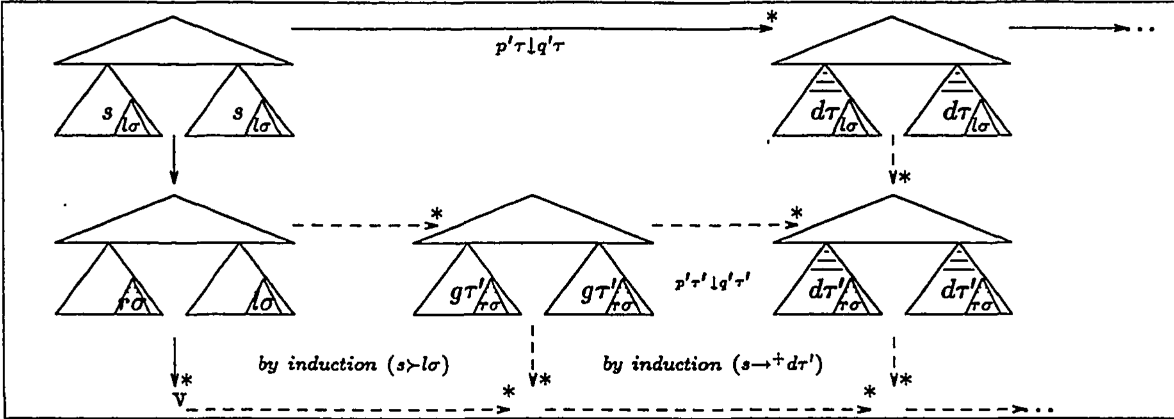

Figure 4.9: Variable overlap below

$c[s']_{\Pi'}$. Now, $u[s']_{\Pi} \rightarrow^* u[l\sigma']_{\Pi''}$, by rewriting any additional occurrences of x in l that were not included in Π. Moreover, $l\sigma'$ rewrites to $r\sigma'$, since, by induction, $p\sigma' \downarrow q\sigma'$ as we show below.

Let *eq* be a new binary operator not appearing in any rule. Consider the derivation $eq(p\sigma, q\sigma) \rightarrow^* eq(w, w)$ (known to exist for some w since $p\sigma \downarrow q\sigma$ in depth $n-1$). We also know that $p\sigma \rightarrow^* p\sigma'$ and $q\sigma \rightarrow^* q\sigma'$, by application of $s \rightarrow s'$ in the substitution parts. By induction on the second component, we have that $eq(p\sigma', q\sigma') \downarrow eq(w, w)$ from which it follows that $p\sigma' \downarrow q\sigma'$ since there are no rules for *eq*. Thus, $u[s']_{\Pi} \rightarrow^* u[r\sigma]_{\Pi''} \downarrow v$. This is illustrated in Figure 4.8.

Similarly, if π is inside some s, we have $u[s']_{\Pi} \rightarrow^* u[d\tau'] \downarrow v$, as shown in Figure 4.9 Here τ' is like the substitution *tau* used to rewrite $s \rightarrow s'$, but maps the variable in g to $c[r\sigma]$ instead of to $c[l\sigma]$. The condition $p'\tau' \downarrow q'\tau'$, needed to show that $u' \rightarrow^* u[g\tau']_{\Pi} \rightarrow u[d\tau']_{\Pi}$, holds by induction on the first component, since $l\sigma$ is a proper subterm of s. □

4.6 Conclusion

We have explored two different restrictions on critical pairs of conditional rewrite systems, namely shallow joinability and overlays only, and proved confluence results for systems meeting those restrictions. Our proofs show that, for conditional systems, the notions of confluence, local-confluence, and joinable critical pairs can not be neatly disentangled. In particular, termination was needed to show that a system is locally confluent even if all critical pairs are shallow joinable. We have also presented counter-examples which show that all our restrictions are necessary.

5 COMPLETION METHODS FOR CONDITIONAL EQUATIONS

In this chapter, we study completion methods for generating a convergent rewrite system equivalent to a set of conditional equations. First, we describe completion methods for unconditional theories. This works by orienting equations into terminating rules, using a well-founded ordering, and generating new equations (rules) by superposing left-hand sides until all critical pairs are joinable. This method only fails when an equation is unorientable (as a rule) in either direction under the well-founded ordering.

For conditional systems, the basic idea is the same—orient equations into rules and generate new rules by superposition between left-hand sides. But we run into some additional problems. The joinability of critical pairs ensures confluence only for decreasing systems. So, whenever we encounter an equation that is not decreasing—this can happen even when the original set of equations is decreasing—the straightforward completion approach fails. We have shown sufficient conditions for confluence of non-decreasing systems (for example, when all critical pairs are overlays or shallow-joinable). But, in the context of completion, it is not desirable to work with non-decreasing rules, because even checking if a rule applies to a term is, in general, undecidable with such rules.

Methods to handle non-decreasing equations, within the completion framework, have been proposed. In [Kaplan 1987] and [Jouannaud and Waldmann, 1986] a technique for detecting some cases when (non-decreasing) critical pairs are infeasible is suggested. This is done by using "narrowing" (equation solving) along with completion. Ganzinger [Ganzinger, 1987] proposed an improvement to handle even feasible, non-decreasing critical pairs by using narrowing to enumerate the solutions for which such critical pairs are feasible. This enables one to replace, in some cases, a non-decreasing

equation by a set of decreasing ones, without losing any consequences of the non-decreasing equation.

We show here how to judiciously use the translation scheme introduced in Chapter 3 to convert non-decreasing conditional equations to decreasing equations in the context of completion using an *if* operator. Operationally, this has an effect similar to Ganzinger's idea of enumerating solutions of critical pairs, because we now have the ability to superpose in terms that were originally in the condition. By using an optimization (critical-pair criterion) while superposing in the rules corresponding to the non-decreasing equations, we show how to capture Ganzinger's method exactly. In this framework, it is also easy to express "contextual simplification" of critical pairs. We illustrate these techniques with an interesting example. In the Appendix, we describe an experimental implementation of completion within the rewrite rule laboratory RRL.

5.1 Unconditional Completion

In this section, we describe completion methods for unconditional equations using inference rules (formulated here as conditional rule schemas) following the abstract approach used in [Bachmair, 1987 ; Dershowitz, 1989].

At any stage of completion we have a set of equations E_i and a set of rules R_i. Initially we start with the input set of equations E_0 and no rules. We are also given a well-founded ordering $\succ$ that can be used to compare terms. A single step of completion may be viewed as applying an inference rule to transform the current pair $\langle E_i \ , \ R_i \rangle$ to an "equivalent" $\langle E_{i+1} \ , \ R_{i+1} \rangle$.

At a given stage, we can choose any applicable inference rule (non-deterministically) leading to different completion sequences for the same set of input equations and orderings. A successful sequence is one that generates $\langle \emptyset \ , \ R_f \rangle$, where the final set of rules R_f is convergent (terminating

and confluent). A failing sequence is one where none of the inference rules can be applied (for example, when we have an equation that is not orientable in either direction) even though we do not have a convergent rewrite system.

The following four basic inference rules capture the essential operations that we use during completion.

Delete: When an equation is trivial (of form $s = s$) it can be deleted.

Simplify: If $s = t$ is an equation in E_i, and s can be rewritten to a term u, using some rule in R_i, then we can replace $s = t$ by $u = t$.

Orient: If $s = t$ is an equation and $s \succ t$ in the given ordering, then this equation can be oriented into the rule $s \rightarrow t$.

Deduce: If $s = t$ is a critical pair in R_i (obtained by superposing left-hand sides of rules—see definition in Chapter 2), then we can add $s = t$ as an equation. We only need to generate a critical pair in R_i that has not been generated in some earlier R_j. That is, each overlap between rules needs to be considered only once.

In practice, it is best to apply the rules in the order given (orienting equations only after simplifying them and deleting trivial ones, and deducing critical pairs last). In addition, two more inference rules are used to keep the rule set simplified as much as possible. These are:

Compose: If the right hand side t of a rule $s \rightarrow t$ can be rewritten to u, then the rule can be replaced by $s \rightarrow u$.

Delete	
	$\langle E \cup \{s=s\}\ ,\ R\rangle \;\Rightarrow\; \langle E\ ,\ R\rangle$
Simplify	
	$s \rightarrow u\ :\ \langle E \cup \{s=t\}\ ,\ R\rangle \;\Rightarrow\; \langle E \cup \{u=t\}\ ,\ R\rangle$
Orient	
	$s \succ t\ :\ \langle E \cup \{s=t\}\ ,\ R\rangle \;\Rightarrow\; \langle E\ ,\ R \cup \{s \rightarrow t\}\rangle$
Deduce	
	$s=t \in \mathit{critpairs}(R)\ :\ \langle E\ ,\ R\rangle \;\Rightarrow\; \langle E \cup \{s=t\}\ ,\ R\rangle$
Compose	
	$t \rightarrow u:\ \langle E\ ,\ R \cup \{s \rightarrow t\}\rangle \;\Rightarrow\; \langle E\ ,\ R \cup \{s \rightarrow u\}\rangle$
Collapse	
$s \triangleright l \;\&\; s \rightarrow u \;\mathit{using}\; l \rightarrow r \in R : \langle E\ ,\ R \cup \{s \rightarrow t\}\rangle \;\Rightarrow\; \langle E \cup \{u=t\}\ ,\ R\rangle$	

Table 5.1: Inference Rules for Unconditional Completion

Collapse: If the left hand side s of a rule $s \rightarrow t$ can be rewritten to u by a rule $l \rightarrow r$ and l is "simpler" (in some sense) than s (for example, if $s \rightarrow t$ cannot reduce l), then we can delete the rule $s \rightarrow t$ and add the equation $u = t$. We use $s \triangleright l$ to denote the "simpler" relation.

These rules are shown in Table 5.1.

A completion procedure based on these rules can be proved correct using *proof orderings* [Bachmair, 1987]. A proof of an equation $s = t$ (with the pair $\langle E_i\ ,\ R_i\rangle$) is a sequence of steps:

$$s - s_1 - \cdots - s_n - t$$

Each step is either an application of an an equation ($\leftrightarrow$) in E_i or a rule (in either direction, $\rightarrow$ or $\leftarrow$) in R_i.

The inference rules are "sound," in the sense that, if $\langle E_i\ ,\ R_i\rangle \Rightarrow \langle E_{i+1}\ ,\ R_{i+1}\rangle$ using some inference rule, then $s = t$ is provable in $\langle E_i\ ,\ R_i\rangle$ (written $\langle E_i\ ,\ R_i\rangle \vdash s = t$) iff $s = t$ is provable

in $\langle E_{i+1}\ ,\ R_{i+1}\rangle$. Moreover, we can choose an appropriate ordering on proofs to show that there is a proof of $s = t$ in $\langle E_{i+1}\ ,\ R_{i+1}\rangle$ that is the same or "smaller" than the one in $\langle E_i\ ,\ R_i\rangle$. That is, the inference rules can make proofs simpler in a well founded ordering. A proof of $s = t$ is in normal form (a rewrite or valley proof) in some $\langle E_j\ ,\ R_j\rangle$ if it is of the form $s \rightarrow^* u \,{}^*\!\leftarrow t$. That is, s and t can be rewritten to a common term u using the rules in R_j.

These inference rules are also "complete" in that, a "fair" application of the inference rules (no critical pair that can be obtained by deduce is indefinitely neglected), will eventually transform any equational proof of $s = t$ in the initial theory E_0, into a rewrite (valley) proof in some $\langle E_j\ ,\ R_j\rangle$, provided we can always orient all equations into rules. In this sense, completion is *refutationally complete*. Unfailing completion (e.g. [Bachmair et al., 1987], [Hsiang and Rusinowitch, 1987]) is an extension to also handle cases when there are unorientable equations.

5.2 Conditional Completion Methods

How can we adapt the completion techniques for unconditional completion to conditional equations? An initial attempt would be to modify the basic inference rules from the unconditional case.

The "delete" rule can be extended to also remove conditional equations of the form

$$\cdots \wedge s = t \wedge \cdots : s = t$$

where the equation in the conclusion also appears in the condition (for if this equation is to be applicable at all, then $s = t$ must already be provable without using this equation).

The "orient" rule must not only ensure that the left-hand side of the rule is "bigger" than the right-hand side in the ordering, but also that the equation is decreasing. Otherwise, critical pair

Delete

$$\langle E \cup \{c : s = s\} \ , \ R\rangle \ \Rightarrow \ \langle E \ , \ R\rangle$$

$$\langle E \cup \{s = t \ \wedge \ c : s = t\} \ , \ R\rangle \ \Rightarrow \ \langle E \ , \ R\rangle$$

Simplify

$$s \rightarrow u \ : \ \langle E \cup \{c : s = t\} \ , \ R\rangle \ \Rightarrow \ \langle E \cup \{c : u = t\} \ , \ R\rangle$$

$$p \rightarrow u \ : \ \langle E \cup \{p = q \ \wedge \ c : s = t\} \ , \ R\rangle \ \Rightarrow \ \langle E \cup \{u = q \ \wedge \ c : u = t\} \ , \ R\rangle$$

Orient

$$s \succ t, c \ : \ \langle E \cup \{c : s = t\} \ , \ R\rangle \ \Rightarrow \ \langle E \ , \ R \cup \{c : s \rightarrow t\}\rangle$$

Deduce

$$c : s = t \in \mathit{critpairs}(R) \ : \ \langle E \ , \ R\rangle \ \Rightarrow \ \langle E \cup \{c : s = t\} \ , \ R\rangle$$

Table 5.2: Basic Inference Rules for Conditional Completion

joinability does not guarantee confluence. Moreover, if we allow non-decreasing rules, even basic notions like rewriting become undecidable [Kaplan, 1987].

The "simplify" rule can be extended to allow us to rewrite terms in the condition, too. The "deduce" rule is only changed to use the definition of critical pair in Chapter 3.

This gives the inference rules shown in Table 5.2.

As long as all equations (and rules) are decreasing, completion can proceed just like in the unconditional case. But, this does not seem to get us very far. As has been observed in [Kaplan, 1987] and [Jouannaud and Waldmann, 1986], even if we start with a set of equations all of which are decreasing, we often encounter critical pairs that are non-decreasing equations, leading to failure of completion.

Consider the following simple example. We start with the equations, which define $f(x)$ to be 0 when x is *odd* and $s(0)$ when x is *even*.

1.		$even(0)$	$=$	tt
2.		$odd(s(0))$	$=$	tt
3.		$even(s(s(x))$	$=$	$even(x)$
4.		$odd(s(s(x))$	$=$	$odd(x)$
5.	$odd(x) = tt$:	$f(x)$	$=$	0
6.	$even(x) = tt$:	$f(x)$	$=$	$s(0)$

With an appropriate choice of ordering, we can show all these equations are decreasing and we can orient them as rules from left to right.

Rule 5 and rule 6 yield the following non-decreasing critical pair:

$$odd(x) = tt \ \wedge \ even(x) = tt : s(0) = 0$$

This is the only critical pair in the system and it is actually infeasible, for no value of x can be both *odd* and *even* in our system. But completion fails at this stage. The enhancement suggested in [Kaplan, 1984] and [Jouannaud and Waldmann, 1986] is to use "narrowing" (explained in Chapter 6) to detect this infeasibility.

5.2.1 Handling Non-Decreasing Equations

We describe, in this section, how we can incorporate the "narrowing" idea into completion itself, by using a translation mechanism. This also extends to handling some feasible non-decreasing equations à la Ganzinger [1987]. We view completion as working over a (conservative) extension of the original theory. Equations that are non-decreasing may be convertible to decreasing ones in the

Translate :		
$\langle E \cup \{p = q \;\wedge\; c : s = t\}\ ,\ R\rangle$	$\Rightarrow$	$\langle E \cup \{c : if(eq(p,q),s) = if(eq(p,q),t)\}\ ,\ R\rangle$
$\langle E \cup \{c : if(eq(p,q),s) = if(eq(p,q),t)\}\ ,\ R\rangle$	$\Rightarrow$	$\langle E \cup \{p = q \;\wedge\; c : s = t\}\ ,\ R\rangle$

Table 5.3: Translation Rules

extension. If completion succeeds over the new vocabulary, then the rewrite system so generated must also be convergent for the original theory.

Recall the translation to unconditional equations suggested in Chapter 3. Let E be a set of conditional equations using terms from $T(F,X)$. Let if, eq and $true$ be function symbols not in F and $F' = F \cup \{if, eq, true\}$. We can convert E to E' over $T(F',X)$ as follows. We add the equations $eq(x,x) = true$ and $if(true,x) = x$ to E'. A conditional equation $p = q : s = t$ in E may be represented by $if(eq(p,q),s) = if(eq(p,q),t)$ in E'.

In general, a conditional equation may have more than one equation in the condition. We generalize this notion of translation to move any equation (or subset of equations) from the condition to the conclusion. That is, we may replace $p = q \wedge c : s = t$ by $c : if(eq(p,q),s) = if(eq(p,q),t)$, where c stands for the rest of the equations in the condition. This translation is also reversible, in that we can replace (whenever needed) an equation $c : if(eq(p,q),s) = if(eq(p,q),t)$ by $p = q \wedge c : s = t$.

As inference rules for completion, these rules are shown in Table 5.3. We assume that $if(true,x) = x$ and $eq(x,x) = true$ are already in E.

The soundness of these translation rules follows from the fact that it is a conservative extension.

Proposition 5.1 *Let u, v be terms in $T(F,X)$ and $\langle E1\ ,\ R\rangle \Rightarrow \langle E2\ ,\ R\rangle$ using translation. Then $u \leftrightarrow^* v$ is provable in $\langle E1\ ,\ R\rangle$ iff it is provable in $\langle E2\ ,\ R\rangle$.*

In the example, the non-decreasing critical pair

$$odd(x) = tt \ \wedge \ even(x) = tt : s(0) = 0$$

may be replaced by a decreasing equivalent (by moving the $even(x) = tt$ to the consequent), and oriented into a decreasing rule (from left to right, if we assume *even* is "bigger" than *odd* in our ordering):

$$7. \ odd(x) = tt : \ if(eq(even(x), tt), s(0)) \rightarrow if(eq(even(x), tt), 0)$$

Proceeding with completion, we find that this new rule generates two critical pairs. One is:

$$odd(s(s(x))) = tt : \ if(eq(even(x), tt), s(0)) = if(eq(even(s(s(x))), tt), 0)$$

with the rule $even(s(s(x))) \rightarrow tt$. This easily simplifies to an instance of rule 7 and can be deleted.

The other critical pair, with $even(0) \rightarrow tt$, is:

$$odd(0) = tt : if(eq(tt, tt), s(0)) = if(eq(even(0), tt), 0)$$

which after simplification, translation and orientation gives the decreasing rule:

$$if(eq(odd(0), tt), s(0)) \rightarrow if(eq(odd(0), tt), 0)$$

Completion stops successfully as there are no further critical pairs and we get the convergent system R_f:

i.			$eq(x,x)$	$\rightarrow$	$true$
ii.			$if(true,x)$	$\rightarrow$	x
1.			$even(0)$	$\rightarrow$	tt
2.			$odd(s(0))$	$\rightarrow$	tt
3.			$even(s(s(x)))$	$\rightarrow$	$even(x)$
4.			$odd(s(s(x)))$	$\rightarrow$	$odd(x)$
5.	$odd(x) \downarrow tt$	:	$f(x)$	$\rightarrow$	0
6.	$even(x) \downarrow tt$	:	$f(x)$	$\rightarrow$	$s(0)$
7.	$odd(x) \downarrow tt$	:	$if(eq(even(x),tt),s(0))$	$\rightarrow$	$if(eq(even(x),tt),0)$
8.			$if(eq(odd(0),tt),s(0))$	$\rightarrow$	$if(eq(odd(0),tt),0)$

R_f is convergent and is a conservative extension of the original equational theory E_0. So, for all terms u, v in the original signature $(T(F,X))$, such that $u \leftrightarrow^* v$ in E_0, there is a rewrite proof $u \rightarrow^* s^* \leftarrow v$ in R_f. In any such rewrite proof, rules i , ii, 7 and 8 (which have if— a function symbol not in F—as the outermost operator of the left-hand side) cannot be used. So these rules can now be dropped and rules 1-6 form a convergent rewrite system to decide the validity problem for E_0.

5.2.2 Contextual Simplification

In this section, we describe *contextual rewriting* (or rewriting with assumptions) for simplifying critical pairs that arise during conditional completion. A similar idea, for hierarchical systems, is

considered in [Zhang and Rémy, 1985]. This not only make the procedure more efficient, but is also crucial to help it terminate successfully in many cases.

We illustrate, first, with an example. Consider the rules:

1.	$member(x,z) \downarrow ff$	:	$delete(x,z)$	$\rightarrow$	z
2.	$different(x,y) \downarrow tt$	:	$delete(x, y \cdot z)$	$\rightarrow$	$y \cdot delete(x,z)$
3.	$different(x,y) \downarrow tt$	:	$member(x, y \cdot z)$	$\rightarrow$	$member(x,z)$

The critical pair

$$different(x,y) = tt \ \wedge \ member(x, y \cdot z) = ff : \ y \cdot delete(x,z) = y \cdot z$$

between the rules 1 and 2, is not simplifiable directly, as no subterm (in the condition or conclusion) is rewritable directly, using the rules 1 to 3.

But we do not need to rewrite the subterm $member(x, y \cdot z)$ in the condition in isolation. When rewriting this, we can assume that the other condition $different(x,y) = tt$ holds. If we do this, then rule 3 does apply and the condition can be rewritten to $different(x,y) = tt \ \wedge \ member(x,z) = ff$.

Once this is done, the subterm $delete(x,z)$ can be rewritten in the new context, which has $member(x,z) = ff$, using rule 1 to z. The critical pair after these two simplifications is now

$$different(x,y) = tt \ \wedge \ member(x,z) = ff \ : \ y \cdot z = y \cdot z$$

which is trivial. Thus, using contextual rewriting, we have shown that this critical pair is actually joinable in the system.

We denote contextual rewriting by $\underset{C}{\rightarrow}$, where C denotes the context. A context is a conjunction of equations, of the form $p_1 = q_1 \ \wedge \cdots \wedge \ p_n = q_n$, which are assumed to "hold" when doing a

Simplification :
$s \underset{c}{\rightarrow} u : \ \langle E \cup \{c : s = t\} \ , \ R\rangle \ \Rightarrow \ \langle E \cup \{c : u = t\} \ , \ R\rangle$
$p \underset{c}{\rightarrow} u : \ \langle E \cup \{p = q \wedge c : s = t\} \ , \ R\rangle \ \Rightarrow \ \langle E \cup \{u = q \wedge c : s = t\} \ , \ R\rangle$

Table 5.4: Contextual Simplification Rules

rewriting. That is, whenever we have to check if a condition holds to apply a rule, we can use these assumptions to prove it.

We define this formally as follows. Let R be a conditional rewrite system. We say that a term s rewrites in the context C to a term s_1 (denoted $s \underset{C}{\rightarrow} s_1$, if one of the following holds:

1. $s \rightarrow s_1$ (with no context), or
2. $s = s_1$ is an instance of an equation $p_i = q_i$ in the context, and $s \succ s_1$ or
3. $u \downarrow v : l \rightarrow r$ is a conditional rule in R, σ is a substitution such that $l\sigma$ is a subterm of s at position π (i.e., $s \mid_\pi = l\sigma$), $u\sigma$ and $v\sigma$ are joinable in R under the same context C (i.e., $u\sigma \overset{*}{\underset{C}{\rightarrow}} t$ and $v\sigma \overset{*}{\underset{C}{\rightarrow}} t$, for some t), and s_1 is $s[r\sigma]_\pi$.

When simplifying a conditional equation, contextual rewriting adds more power. When rewriting a subterm in the consequent, the context is all the equations in the condition. When rewriting a subterm in the condition, the other equations in the condition form the context. Table 5.4 gives the inference rules for contextual simplification.

That these simplifications are sound is expressed in the following Proposition.

Proposition 5.2 *Let $\langle E_i \ , \ R_i\rangle$ be transformable to $\langle E_{i+1} \ , \ R_{i+1}\rangle$ using contextual simplification. Then $\langle E_i \ , \ R_i\rangle \ \vdash \ s \leftrightarrow^* t$ iff $\langle E_{i+1} \ , \ R_{i+1}\rangle \ \vdash \ s \leftrightarrow^* t$*

The example in Section 5.2.4 illustrates the power of contextual rewriting for simplification of non-decreasing equations.

5.2.3 Critical Pair Optimization

In this section, we present an important optimization of completion that is sound and helps the procedure to terminate more often. This optimization is a restriction of critical pairs involving rules of the form $c : if(c_1, l) \rightarrow if(c_1, r)$ which are obtained by using the *Translate* inference rule.

We motivate the presentation with an example. Consider the equations E over terms $T(F, X)$ (where $F = \{odd, even, s, 0, f, g, h\}$) below.

1.			$even(0)$	$=$	tt
2.			$odd(s(0))$	$=$	tt
3.			$even(s(s(x))$	$=$	$even(x)$
4.			$odd(s(s(x))$	$=$	$odd(x)$
5.			$g(h(y))$	$=$	$g(y)$
6.	$odd(x) = tt$	:	$f(x, y)$	$=$	$g(y)$
7.	$even(x) = tt$	:	$f(x, y)$	$=$	y

They can all be oriented as decreasing rules from left to right (assuming an appropriate ordering). Rules 6 and 7 yield a non-decreasing critical pair:

$$odd(x) = tt \ \wedge \ even(x) = tt : g(y) = y$$

As in the earlier example, this critical pair is actually infeasible, as the condition does not hold for any substitution. The Translate rule allows us to replace this by:

$$8. \ \ odd(x) \downarrow tt : if(eq(even(x), tt), g(y)) \rightarrow if(eq(even(x), tt), y)$$

We now are working over the vocabulary $T(F \cup \{if, eq, \&, true\}, X)$ and assume that rules $eq(x, x) \rightarrow true$ and $if(true, x) \rightarrow x$ also exist.

The superposition in the subterm $even(x)$ of the left-hand side, yields two critical pairs (as in the earlier example). One gives the rule:

$$9. \quad if(eq(odd(0), tt), g(y)) \rightarrow if(eq(odd(0), tt), y)$$

and the other can be simplified to a trivial equation.

If we could stop now, then rules 1-7 form a convergent rewrite system for E. But the fifth rule $g(h(x)) \rightarrow g(y)$ also overlaps with the $g(y)$ subterm in rule 9 (and rule 8), yielding new (and useless) rules of the form:

$$if(eq(odd(0), tt), g(y)) \rightarrow if(eq(odd(0), tt), h(y))$$

and

$$if(eq(odd(0), tt), g(y)) \rightarrow if(eq(odd(0), tt), h(h(y)))$$

And this process goes on, yielding infinitely many rules of this pattern, none of which are useful in proving any equality between terms in $T(F, X)$, because the condition is never feasible. This suggests that in rules of the form $c : if(c_1, l) \rightarrow if(c_1, r)$ it is sufficient to consider superpositions only in c_1.

The main idea in proving that this restriction on critical pairs is complete, is that if the rule $c : if(c_1, l) \rightarrow if(c_1, r)$ is ever used in any proof of equality between terms s and t in $T(F, X)$, then it must be used with a substitution σ for which $c_1\sigma \rightarrow^* true$. Otherwise, the if operator will persist in the proof. Hence, it must be the case that $if(c_1\sigma, l\sigma) \rightarrow^* if(true, l\sigma) \rightarrow l\sigma$.

In any such proof, therefore, there must be peak of the form:

$$l\sigma^* \leftarrow if(c_1\sigma, l\sigma) \rightarrow if(c_1\sigma, r\sigma)$$

where the rewriting in $c_1\sigma$ is a critical peak. If all critical peaks inside c_1 are joinable in R, then this proof can be made "smaller" using some extension of the decreasing ordering of R to an ordering on proofs. See [Ganzinger, 1987] for a detailed proof of this claim (although in a slightly different formulation).

5.2.4 An Interesting Example

In this section, we work out in detail an interesting example that illustrates the optimizations we described.

We start with the following definition of $<$ on natural numbers.

1.		$0 < s(0)$	$=$	tt
2.		$s(x) < s(y)$	$=$	$x < y$
3.	$x < y = tt \ \wedge \ y < z = tt$:	$x < z$	$=$	tt

The first two equations are orientable as decreasing rules from left-to-right. The third equation (transitivity) is not decreasing. It has an "extra" variable y in the condition that is not in either side of the conclusion. Translating yields the following rule:

$$3. \ \ y < z \downarrow tt: \quad if(eq(x < y, tt), x < z) \rightarrow if(eq(x < y, tt), tt)$$

In this rule, we will only superpose in the $eq(x < y, tt)$ subterm of the left-hand side following the optimization of the previous section. The non-decreasing critical pair with rule 1 ($0 < s(0) \rightarrow tt$) is:

$$s(0) < z = tt: \quad if(eq(tt, tt), 0 < z) = if(eq(0 < s(0), tt), tt)$$

After simplification and translation this yields the rule:

$$4. \ \ if(eq(s(0) < z, tt), 0 < z) \rightarrow if(eq(s(0) < z, tt), tt)$$

The critical pair between rules 2 and 3 is:

$$s(y) < z: \quad if(eq(x < y, tt), s(x) < z) \rightarrow if(eq(s(x) < s(y), tt), tt)$$

After simplification and two translations (first to move the $eq(x < y, tt)$ term to the condition, and then to move $s(y) < z$ to the conclusion—we do this for pragmatic reasons) we get the rule:

$$5. \ x < y \downarrow tt: \quad if(eq(s(y) < z, tt), s(x) < z) \rightarrow if(eq(s(y) < z, tt), tt)$$

Rule 4 has only one critical pair (with rule 2):

$$if(eq(0 < z, tt), 0 < s(z)) = if(eq(s(0) < s(z), tt), tt)$$

Simplifying and translating (move $eq(0 < z, tt)$ back to the condition) we get the following decreasing conditional rule:

$$6. \ 0 < z \downarrow tt: \quad 0 < s(z) \rightarrow tt$$

This translation back to a conditional rule is essential to the termination of the completion procedure, for we prevent further superposition in the $0 < z$ term, which now appears in the condition.

Rules 5 and 2 yield the following critical pair:

$$x < y = tt: \quad if(eq(y < z, tt), s(x) < s(z)) = if(eq(s(y) < s(z), tt), tt)$$

Simplifying and translating this to $y < z = tt : if(eq(x < y, tt), x < z) = if(eq(x < y, tt), tt)$ makes the $if(eq(x < y, tt), x < z)$ rewritable in the context $y < z = tt$ by rule 3. This yields a trivial equation and this critical pair is, therefore, joinable.

Rule 6 (the new decreasing rule) has the following critical pair with rule 1

$$0 < 0 = tt: \quad tt = tt$$

which is trivial, and with rule 3 we have:

$$0 < y = tt \ \wedge \ s(y) < z = tt : if(eq(tt, tt), 0 < z) = if(eq(0 < s(y), tt), tt)$$

This critical pair is also joinable. First, we rewrite $0 < s(y)$ to tt in the context $\{0 < y = tt \wedge s(y) < z = tt\}$ using rule 6. Further simplification by $eq(x,x) \rightarrow true$ and $if(true, x) \rightarrow x$ yields:

$$0 < y = tt \ \wedge \ s(y) < z = tt : 0 < z = tt$$

which can be translated to:

$$0 < y = tt : \ if(eq(s(y) < z, tt), 0 < z) = if(eq(s(y) < z, tt), tt)$$

This is an instance of rule 5 and hence, can be simplified contextually to a trivial equation.

The completion procedure now halts, with the following rules:

1.			$0 < s(0)$	$\rightarrow$	tt
2.			$s(x) < s(y)$	$\rightarrow$	$x < y$
3.	$y < z \downarrow tt$	:	$if(eq(x < y, tt), x < z)$	$\rightarrow$	$if(eq(x < y, tt), tt)$
4.			$if(eq(s(0) < z, tt), 0 < z)$	$\rightarrow$	$if(eq(s(0) < z, tt), tt)$
5.	$x < y \downarrow tt$	:	$if(eq(s(y) < z, tt), x < z)$	$\rightarrow$	$if(eq(s(y) < z, tt), tt)$
6.			$0 < z \downarrow tt$	$\rightarrow$	$0 < s(z)tt$

Dropping the if-rules we get the following 3 rule system which is convergent and equivalent to the original equations over $T(\{<, s, 0, tt\}, X)$.

1.			$0 < s(0)$	$\rightarrow$	tt
2.			$s(x) < s(y)$	$\rightarrow$	$x < y$
3.	$0 < z \downarrow tt$	:	$0 < s(z)$	$\rightarrow$	tt

5.3 Conclusion

We have formulated a completion method for conditional equations using sound inference rules and examined contextual rewriting and an optimization (critical pair criteria) to help a procedure based on these rules to terminate in more cases. If completion terminates, we get a convergent rewrite system equivalent to the original equations.

It remains to show that the inference rules for completion can also be used as a semi-decision procedure for validity. For this, we have to extend the proof ordering techniques used for unconditional equations and show that a "fair" application of the inference rules can transform any equational proof to a rewrite proof. In [Ganzinger, 1987] this is addressed, in a slightly different framework that does not use the translation mechanism, but works directly with non-decreasing equations. See also [Kounalis and Rusinowitch, 1988] for a different technique for solving the word problem in Horn theories.

6 EQUATION SOLVING METHODS

In this chapter, we address the problem of solving equations in (conditional) theories. We first introduce the paradigm of equational programming. We then examine two methods that have been proposed for solving equations—*narrowing* and *decomposition*—and point out their drawbacks. Finally, we describe a *goal-directed* equation solving algorithm, formulated using conditional rules, which combines nice features of top-down decomposition and narrowing and also has pruning rules that enable it to search for solutions more efficiently and halt more often when equations are infeasible.

6.1 Equational Programming

Equational programming using unconditional equations to capture Lisp-like applicative programming is studied in detail in [O'Donnell, 1985]. Later, several proposed programming languages use (conditional) equations as a means of extending equational programming by capturing the simple syntax and semantics of Prolog-like logic programming languages; such languages include RITE [Dershowitz and Plaisted, 1985], SLOG [Fribourg, 1985], and EQLOG [Goguen and Meseguer, 1986].

A program, in this paradigm, is a set of directed (conditional) equations (rules). Computing consists of finding values (substitutions) for the variables in a goal $s = t$ for which the terms are provably equal. Consider the following example of a system for reversing a list, where rev is reverse

and *tcons* adds an element to the end of a list.

$$\begin{array}{rcl} rev(nil) & \rightarrow & nil \\ rev(A \cdot X) & \rightarrow & tcons(rev(X), A) \\ tcons(nil, A) & \rightarrow & A \cdot nil \\ tcons(B \cdot X, A) & \rightarrow & B \cdot tcons(X, A) \end{array}$$

A goal of the form $X = rev(1 \cdot 2 \cdot nil)$ can be solved by *rewriting* the right-hand side of the goal to yield $X = 2 \cdot 1 \cdot nil$; rewriting corresponds to the functional part of equational programming. On the other hand, a query like $rev(X) = 1 \cdot 2 \cdot nil$, requires *equation solving* to produce the value(s) for X that satisfies the equation. This query has the answer, $\{X \mapsto 2 \cdot 1 \cdot nil\}$. Finding solutions corresponds to the logic programming capability.

As an example using conditional equations, consider the following program which defines an insertion sort:

$$\begin{array}{rrrcl} & & isort(nil) & \rightarrow & nil \\ & & isort(X \cdot Y) & \rightarrow & insert(X, isort(Y)) \\ & & insert(X, nil) & \rightarrow & X \cdot nil \\ X < Y = tt & : & insert(X, Y \cdot Z) & \rightarrow & X \cdot (Y \cdot Z) \\ X < Y = ff & : & insert(X, Y \cdot Z) & \rightarrow & Y \cdot insert(X, Z) \end{array}$$

where $<$ can also be defined using conditional equations, as in the previous chapter.

Equational languages have more expressive power than Prolog because they have both functionality and built-in equality. One can also incorporate streams and destructive assignments. Illustrative examples can be found in [Dershowitz and Plaisted, 1988].

Any pure Prolog program can be translated into a set of conditional equations (or rewrite rules) by using a distinguished constant *true*. A unit clause p is replaced by the rule

$$p \rightarrow true$$

A clause of the form $l : -p_1, \ldots, p_n$ can be replaced by

$$p_1 = true \land \ldots \land p_n = true \;:\; l \;\rightarrow\; true$$

Solving a goal g in Prolog can be viewed as the satisfiability problem for the equation $g = true$. Narrowing to solve this goal is similar to Prolog's computation strategy to solve the goal.

This simple translation raises several issues. While the set of rules is terminating, it is not decreasing, as terms in the condition often have variables that are not in the left-hand side. But for such programs (obtained from pure Prolog clauses) we do have *confluence* since predicates can only be rewritten to *true* and also do not occur as subterms in any goal. For general equational programs, however, it is more difficult to show confluence.

Solving equations is the basic operation in interpreters for such equational languages and efficient methods are of critical importance. In general, paramodulation can be used (as in resolution-based theorem provers) to solve equations, but is highly inefficient. For equational theories that can be presented as a (ground) confluent rewrite system, better equation-solving methods have been devised, *narrowing* [Slagle, 1974; Fay, 1979; Hullot, 1980] being the most popular.

6.2 Procedures for Equation Solving

An *equational goal* is written in the form $s \downarrow_? t$, where s and t are, in general, terms containing variables. A *solution* to such a goal is a substitution σ such that $s\sigma \downarrow t\sigma$. This means that $s\sigma$ is equal to $t\sigma$, in the underlying theory, for all substitutions of terms for variables in $s\sigma$ and $t\sigma$. A

solution is *irreducible* if each of the terms substituted for the variables in the goal are irreducible. Note that the terms s and t are interchangeable, since $s \downarrow_? t$ iff $t \downarrow_? s$; in this sense, equational goals are *unoriented.*

An equation solving procedure is *complete* if it can produce all solutions to any goal, up to equality in the underlying theory. That is, if σ is a solution to $s \downarrow_? t$, then a complete procedure will produce a solution μ for the goal that is at least as general as σ. The more general a solution, the smaller it is under the following (quasi-) ordering $\preceq$ on substitutions: $\mu \preceq \sigma$ iff there exists a substitution τ such that $(X\mu)\tau \leftrightarrow^* X\sigma$, for all variables X (where $\leftrightarrow^*$ is the reflexive, symmetric, and transitive closure of $\rightarrow$). For example, if R is just the rule $\{0 + U \rightarrow U\}$, the solution $\{X \mapsto 0, Y \mapsto 0\}$ to the goal $X + Y \downarrow_? 0$ is more general than the (reducible) solution $\{X \mapsto 0 + 0, Y \mapsto 0 + 0\}$.

For clarity and uniformity of presentation, we will formulate all the equation solving procedures using inference rules. We will assume that we have a set of goals to solve simultaneously (initially the singleton set of the input goal—$\{s \downarrow_? t\}$). Inference rules can be used to transform the current set of equational goals into a new set of goals in a sound manner, i.e. a solution to the new set of goals will also be a solution to the original set. A successful solution is reached when the initial set of goals can be transformed to the empty set. The inference rules themselves are represented as conditional rules (schemas). Each transformation step may introduce substitutions and the composition of all these is the solution to the original goal. In our presentation, we will not clutter the rules with information on how to keep track of the solution (composing intermediate substitutions).

6.2.1 The Narrowing Procedure

Narrowing uses unification (instead of matching) to "apply" rules to terms that may contain variables. Since rule variables are universally quantified, one can always rename them so that the rule and term have no variable in common. For example, if $U + 0 \rightarrow U$ is a rule, then $(X+Y)+Z$ narrows to $X + Z$ via substitution $\{Y \mapsto 0\}$. For conditional rules, the unifying substitution, between the left hand side and the non-variable subterm to be narrowed, must first be extended to be feasible for the equations in the condition. This leads to a recursive definition of conditional narrowing.

Formally, a term s is said to *narrow* to a term t (via a substitution σ), symbolized $s \overset{\sigma}{\rightsquigarrow} t$, if s contains a nonvariable subterm $s|_p$ that unifies, via most general unifier μ, with the left-hand side l of a rule $c_1 \downarrow c_2 : l \rightarrow r$ (whose variables have been renamed so that they are distinct from those in s), τ is a substitution such that $c_1\mu\tau \downarrow c_2\mu\tau$, $\sigma = \mu \circ \tau$ and $t = s\sigma[r\sigma]_p$. In effect, we apply the substitution σ to t and then rewrite the subterm of $s\sigma$ using $c_1 \downarrow c_2 : l \rightarrow r$.

To illustrate the above definition, let R be the rules:

$$\begin{array}{rcrcl} & & f(a,b) & \rightarrow & a \\ f(x,y)\downarrow a & : & g(a,y) & \rightarrow & y \end{array}$$

The term $h(g(U,V),V) \overset{\{U \mapsto a, V \mapsto b\}}{\rightsquigarrow} h(b,b)$ since $g(U,V)$ unifies with the left hand side $g(a,y)$ with the substitution $\mu = \{U \mapsto a, V \mapsto y\}$, which can be extended to be feasible for the condition $f(x,y) \downarrow a$.

To solve goals using the narrowing method, given a confluent system R, two operations are applied to a goal:

Reflect: If σ is the most general unifier of s and t, then σ is a solution of $s \downarrow_? t$.

$\sigma = mgu(s,t) \ : \ \{s \downarrow_? t\} \cup E \ \Rightarrow \ E\sigma$
$c_1 \downarrow c_2 : l \to r \in R \ \& \ \mu = mgu(l, s\mid_p) \ :$ $\{s \downarrow_? t\} \cup E \ \Rightarrow \ \{s[r]_p\mu \downarrow_? t\mu\} \cup \{c_1\mu \downarrow_? c_2\mu\} \cup E\mu$

Table 6.1: Inference Rules for the Narrowing Procedure

Narrow: If $s \overset{\sigma}{\leadsto} s'$, then $s \downarrow_? t$ has a solution if $s' \downarrow_? t\sigma$ does.

A goal for which neither narrowing nor reflection applies is *unsatisfiable*.

The two operations on goals can be formulated as inference rules as shown in Table 6.1 (we use $mgu(s,t)$ to denote a most general unifier of terms s and t). The second rule makes use of the fact that the narrowing operation can be broken down into two parts— unifying the left-hand side l of a rule with a non-variable sunterm s/p and extending the substitution by (recursively) solving the condition.

For example, if R is $\{f(x,x) \to c(x), a \to b\}$, then an input goal set $\{f(a,y) \downarrow_? f(y,b)\}$ "narrows" (using the second rule) to $\{c(a) \downarrow_? f(a,b)\}$ which narrows (reduces, actually) in three steps to $\{c(b) \downarrow_? c(b)\}$. Now, "reflecting" yields the solution $\{y \mapsto a\}$.

6.2.2 Drawbacks of Narrowing

Narrowing can simulate any rewriting strategy (top-down, bottom-up, etc.); hence, it often produces duplicate solutions. For completeness, it is sufficient to simulate any one rewriting strategy. Our goal-directed method—presented in the next section—simulates innermost rewriting.

Quite often, narrowing cannot detect that a goal is unsatisfiable. If we solve the goal $rev(Y) = 1 \cdot nil$ using narrowing, we get the solution $\{Y \mapsto 1 \cdot nil\}$. But the narrowing procedure does not

halt after producing this unique solution. It generates infinitely many failing subgoals of the form $rev(tcons(tcons(\ldots(rev(Y_n), A)\ldots))) = 1 \cdot nil$.

Let us now examine a simple unsatisfiable example, where narrowing does not halt, to motivate our use of *reachability* later. Consider the following system:

$a(f(X))$	$\rightarrow$	$a(X)$
$b(f(X))$	$\rightarrow$	$b(X)$

It is clear that the goal $a(Y) \downarrow_? b(Y)$ is unsatisfiable, because for any substitution for Y any term derived from $a(Y)$ will have a as its outermost operator, while any term derived from $b(Y)$ will have b as its outermost operator. But using narrowing we will never stop as $a(Y) \rightsquigarrow a(Y_1) \rightsquigarrow a(Y_2)\ldots$ and we keep trying to solve new instances of the same equation.

This particular example can be handled by the use of *subsumption* checking as described in [Rety et. al-85]. In general though, the subsumption check cannot solve all the problems caused by these infinitely narrowable terms as shown in the following example. Let R be:

$a(d(x))$	$\rightarrow$	$b(x, x)$
$a(f(e))$	$\rightarrow$	$f(e)$
$b(f(x), y)$	$\rightarrow$	$b(x, f(y))$

and the goal to solve, $a(f(u)) \downarrow_? b(v, e)$. Narrowing will produce infinitely many non-subsuming equations by narrowing the $b(v, e)$ term.

Simple restrictions on narrowing, like narrowing only at the innermost narrowable positions, are incomplete (innermost narrowing does not simulate every possible innermost rewriting). For example, if R is:

$f(x, a)$	$\rightarrow$	0
$g(b)$	$\rightarrow$	0

To solve $f(g(u),u) \downarrow_? 0$, if we narrow only the innermost narrowable subterm $g(u)$ we stop without solution.

Variations on narrowing include: *normal narrowing* (in which terms are normalized via $\xrightarrow{!}$ before narrowing) [Fay, 1979], *basic narrowing* A basic position is a nonvariable position of the original goal or one that was introduced into the goal by the nonvariable part of a right-hand side of the rule applied in a preceding narrowing step. [Hullot, 1980], and their combination [Réty, 1987], all of which are semi-complete for convergent R. In [Bosco, *et al.*, 1987], a strategy derived from simulating SLD-resolution on flattened equations is considered. For a comprehensive treatment of narrowing and E-unification, see [Kirchner, 1985].

6.2.3 The Decomposition Procedure

Using narrowing, one has no control over which (nonvariable) narrowable subterm is used produce new subgoals; all possibilities are explored. Martelli, *et al.* [1986] give a top-down equation-solving procedure, which ignores some narrowings, reducing the search space thereby. There are four basic operations:

Decompose: A goal of the form $f(u_1, ..., u_n) \downarrow_? f(v_1, ..., v_n)$ (with both terms having the same outermost operators), has a solution, if the n subgoals, $u_1 \downarrow_? v_1, \ldots, u_n \downarrow_? v_n$, can be solved simultaneously.

Restructure: A goal $f(u_1, ..., u_n) \downarrow_? t$ has a solution, if $c_1 \downarrow c_2 : f(l_1, ..., l_n) \rightarrow r$ is a rule in R (the left-hand side of which has the same outermost operator as one side of the goal), and the $n + 2$ subgoals, $l_1 \downarrow_? u_1, ..., l_n \downarrow_? u_n, c_1 \downarrow_? c_2$, and $r \downarrow_? t$, can be solved simultaneously.

$$\{f(u_1,...,u_n) \downarrow_? f(v_1,...,v_n)\} \cup E \;\Rightarrow\; \{u_1 \downarrow_? v_1,\ldots,u_n \downarrow_? v_n\} \cup E$$

$$c_1 \downarrow c_2 : f(l_1,...,l_n) \to r \in R \;:\; \{f(u_1,...,u_n) \downarrow_? t\} \cup E \;\Rightarrow\; \{l_1 \downarrow_? u_1,...,l_n \downarrow_? u_n, c_1 \downarrow_? c_2, r \downarrow_? t\} \cup E$$

$$\sigma = mgu(X,t) \;:\; \{X \downarrow_? t\} \cup E \;\Rightarrow\; E\sigma$$

$$occurs(X,t) \;\&\; t|_p = f(t_1,\ldots,t_n) \;\&\; c_1 \downarrow c_2 : f(l_1,...,l_n) \to r \in R \;:\; \{X \downarrow_? t\} \cup E \;\Rightarrow\; \{X \downarrow_? t[r]_p, l_1 \downarrow_? u_1,...,l_n \downarrow_? u_n, c_1 \downarrow_? c_2\} \cup E\}$$

Table 6.2: Inference Rules for the Decomposition Procedure

Bind: If the goal is of the form $X \downarrow_? t$, where X is a variable, and X unifies with t, then $\{X \mapsto t\}$ is a solution.

Expand: If the goal is of the form $X \downarrow_? t$, where X is a variable, but X does not unify with t (because X occurs in t), then it has a solution if the $n+2$ subgoals, $l_1 \downarrow_? t_1,\ldots,l_n \downarrow_? t_n, c_1 \downarrow_? c_2$, and $X \downarrow_? t[r]$, can be solved simultaneously, where $f(t_1,\ldots,t_n)$ is any subterm of t, $f(l_1,...,l_n) \to r$ is a rule in R (with the same outermost operator), and $t[r]$ is t with $f(t_1,\ldots,t_n)$ replaced by r.

As inference rules these are shown in Table 6.2.

A successful application of expansion amounts to narrowing t. The rule $g(f(a)) \to a$ and goal $\{X \downarrow_? f(g(X))\}$ [Martelli, *et al.*, 1986] demonstrates the need for expansion (what they call "full rewriting") in the "occur check" case. Here, we can neither bind nor restructure, but by expanding at the subterm $g(X)$, a solution $\{X \mapsto f(a)\}$ is obtained.

6.2.4 Drawbacks of Decomposition

Though the decomposition method limits the search for solutions, where there are conflicting "constructor" symbols in the goal (a *constructor* is a symbol which is not outermost in any left-hand

side), it introduces some new problems. Consider, for example:

$$\begin{array}{rcl} f(a(X), b(X)) & \to & X \\ f(X, X) & \to & X \\ a(e) & \to & e \\ b(e) & \to & e \end{array}$$

To solve $\{f(e, Y) \downarrow_? e\}$, narrowing would only use the second rule $f(X, X) \to X$, giving the irreducible solution $\{Y \mapsto e\}$. But the decomposition procedure also restructures using the first rule $f(a(X), b(X)) \to X$, to get the new goals: $\{a(X) \downarrow_? e,\ b(X) \downarrow_? Y,\ X \downarrow_? e\}$; this gives another correct, but reducible, solution $\{Y \mapsto b(e)\}$. Thus, decomposition does not take full advantage of the fact that there is no way for e to rewrite to an instance of $a(X)$ that enables the first rule to apply.

Moreover, there are unsatisfiable cases for which narrowing terminates with failure, but decomposition does not halt, as illustrated by the following example:

$$\begin{array}{rcl} f(a(X), b(X)) & \to & a(X) \\ a(d(X)) & \to & a(X) \\ b(d(X)) & \to & b(X) \end{array}$$

Consider solving the goal $f(Y, Y) \downarrow_? Y$. Were one to try and narrow this, the search would stop immediately, as neither term is narrowable. The decomposition procedure, on the other hand, restructures the goal into $\{Y \downarrow_? a(X),\ Y \downarrow_? b(X)\}$, which in turn leads to attempts to solve $\{a(X) \downarrow_? b(X)\}$, with neither success nor failure. By using oriented goals, we show how to combine the advantages of this top-down approach with the elimination of narrowing subterms of left-hand sides.

6.3 Goal Directed Equation Solving

In this section, we introduce two concepts— "oriented goals" and "operator derivability"— both of which are useful for pruning the search for solutions. We use these to formulate a goal-directed equation solving procedure, also using conditional rules/schemas as for narrowing and decomposition. The rules we give form a complete equational program to simulate innermost rewriting sequences.

There is one important difference, however, between the inference rules in this section and those for narrowing and decomposition. Whereas the inference rules were only used to "rewrite" goals to new ones before, here the inference rules are themselves used as an equational program for solving equations and used to "narrow" goals (examples will illustrate this later). This also allows us to eliminate the "bind" and "expand" rules in the decomposition procedure. In a sense, the rules we give act as a meta-circular interpreter for equation solving.

6.3.1 Innermost Rewriting Sequences and Oriented Goals

For convergent rewrite systems, every term has a unique normal form and any rewriting strategy can be used to find it. This leads equation solving procedures like narrowing to duplicate solutions to a goal $s \downarrow_? t$, by faithfully following all rewriting paths that prove that $s\sigma \downarrow t\sigma$. As we stated earlier, simple restrictions on narrowing strategies prove to be incomplete. We choose one complete rewriting strategy (innermost) below and show how we can simulate it by using inference rules similar to those in the decomposition procedure.

A derivation sequence:

$$t = f(t_1, \ldots, t_n) \rightarrow t1 \rightarrow t2 \cdots \rightarrow s$$

is said to be a bottom-up (innermost) rewriting sequence if whenever a rule is applied at some position, no lower position is rewritable. This is similar to the innermost evaluation strategy used in functional languages like Lisp (evaluate the arguments and then apply the function).

An *oriented* goal $t \rightarrow^? s$ has a solution σ if there is an innermost derivation sequence $t\sigma \rightarrow \cdots \rightarrow s\sigma$. Unlike equational goals $t \downarrow_? s$ (which are symmetric in s and t), here we allowing rewritings only in $t\sigma$.

6.3.2 Simulating Innermost Rewriting

Innermost derivations can be classified into two cases, depending on whether or not they contain a rewrite step at the outermost, root position:

Directed Decompose: If no rewrite step ever occurs at the top-level (root) operator (f) of t, then s also must have f as its top operator. That is, $s \equiv f(s_1, \ldots, s_n)$ and there is a bottom-up derivation sequence of the s_i from the t_i.

Directed Restructure: Suppose one rewrite step does take place at the top. Then, the instance of the rule of R first applied at the top must be of the form $c_1 \downarrow c_2 : f(l_1, \ldots, l_n) \rightarrow r$ (with the same outermost operator f as the starting term t) and the subterms t_i of t must have been rewritten to make this rule applicable.

The two inference rules (shown in Table 6.3) constitute a complete *equational* program that solves goals of the form $s \rightarrow^? t$, (where s and t are (first-order) terms that may contain free, "logic" variables). Such goals are solved by finding substitutions σ (for those variables) such that there is an innermost derivation $s\sigma \rightarrow t\sigma$. For convergent systems, equational goals can be easily re-

$\{f(u_1,...,u_n) \rightarrow^? f(v_1,...,v_n)\} \cup E$	$\Rightarrow$	$\{u_1 \rightarrow^? v_1, \ldots, u_n \rightarrow^? v_n\} \cup E$
$c_1 \downarrow c_2 : f(l_1,...,l_n) \rightarrow r \in R$: $\{f(u_1,...,u_n) \rightarrow^? t\} \cup E$	$\Rightarrow$	$\{u_1 \rightarrow^? l_1,...,u_n \rightarrow^? l_n, c_1 \rightarrow^? Z, c_2 \rightarrow^? Z, r \rightarrow^? t\} \cup E$

Table 6.3: Inference Rules for Simulating Innermost Derivations

expressed using oriented goals: replace $\{s \downarrow_? t\}$ by $\{s \rightarrow^? Z, t \rightarrow^? Z\}$, where Z is a new variable. Thus, this can be used as a complete equation solving procedure for convergent rewrite systems.

We now illustrate some advantages of this formulation. Consider again the example:

$f(a(X), b(X))$	$\rightarrow$	X
$f(X, X)$	$\rightarrow$	X
$a(e)$	$\rightarrow$	e
$b(e)$	$\rightarrow$	e

To solve $f(e, Y) \downarrow_? e$, we first replace it by the oriented goals $\{f(e, Y) \rightarrow^? Z, e \rightarrow^? Z\}$. The directed decompose rule succeeds with the second goal and binds Z to e, leaving the subgoal $\{f(e, Y) \rightarrow^? e\}$. The directed restructuring rule for f matches the new subgoal, and either of the two rules in the above system with f as the root operator of the left-hand side match the condition.

If we pick $f(X, X) \rightarrow X$ we get subgoals $\{e \rightarrow^? X, Y \rightarrow^? X\}$, which have a solution $\{Y \mapsto e, X \mapsto e\}$, obtained by decomposition. For the other f rule, $f(a(X), b(X)) \rightarrow X$, the remainder of the condition fails, there being no way to solve $e \rightarrow^? a(X)$. The one successful solution, viz. $\{Y \mapsto e\}$, corresponds to the innermost derivation $f(e, e) \rightarrow^* e$.

Note that no special rules (like *expand*) for the "occur-check" case are necessary. Consider solving the goal $\{g(f(X)) \rightarrow^? X\}$ with rule $f(g(a)) \rightarrow a$. The decompose rule instantiates X to

$g(Z)$ and produces the subgoal $\{f(g(Z)) \rightarrow^? Z\}$, which can be solved by restructuring, yielding $\{X \mapsto g(a)\}$ as a solution.

6.3.3 Operator Rewriting

Our two rule schema serves as the basis of the goal-directed equation-solving procedure. Other than its simplicity, the main advantage of this formulation is that it allows one to easily incorporate additional rules that simplify and prune goals with no loss of completeness. We consider one such technique here which uses operator rewriting.

Let R be a rewrite system over terms constructed from a set $\mathcal{F}$ of function symbols. We consider a derived rewrite system F over $\mathcal{F}$, as follows: For each rule $f(t_1,\ldots,t_n) \rightarrow g(s_1,\ldots,s_m)$ in R (ignoring the terms in the condition), with $f \neq g$, we add a rule $f \rightarrow g$ to F. For each rule $f(t_1,\ldots,t_n) \rightarrow X$ in R, where X is a variable (sometimes referred to as a "collapsing" rule), we add rules $f \rightarrow g_i$ to F for all function symbols g_i other than f in $\mathcal{F}$.

Let f and g be two operators in $\mathcal{F}$. Operator g is *derivable* from f if $f \rightarrow^* g$ in F. This (decidable) notion allows us to prune subgoals during equation solving, since a goal $f(t_1,\ldots,t_n) \rightarrow^? g(s_1,\ldots,s_m)$ is satisfiable in R only if g is derivable from f.

For the reverse example of Section 1 we have:

R			F
$rev(nil)$	$\rightarrow$	nil	$rev \rightarrow nil$
$rev(A \cdot X)$	$\rightarrow$	$tcons(rev(X), A)$	$rev \rightarrow tcons$
$tcons(nil, A)$	$\rightarrow$	$A \cdot nil$	$tcons \rightarrow \cdot$
$tcons(B \cdot X, A)$	$\rightarrow$	$B \cdot tcons(X, A)$	$tcons \rightarrow \cdot$

```
rite(X,Y)  :-  var(X),  !,  unify(X, Y) .
rite(X,Y)  :-  not(derivable(X,Y)),  ! ,  fail.
rite(X,Y)  :-  functor(X,F,N),  functor(Y,F,N),  rites(N,X,Y).
               /* directed  decompose */
rite(X,Y)  :-  functor(X,F,N),  functor(L,F,N),  rule(L,R),
               rites(N,X,L),  rite(R,Y).
               /* directed  restructure */
rites(I,X,Y) :- arg(I,X,Xi),  arg(I,Y,Yi),  rite(Xi,Yi),
                I1  is  I-1,  rites(I1,X,Y).
rites(0,X,Y).
```

Table 6.4: Prolog program for Goal-Directed Equation Solving

Operator *nil* is derivable from *rev* but not from *tcons*. Directed goals of the form $f(t_1, \ldots, t_n) \rightarrow^? g(s_1, \ldots, s_m)$, whose outermost operators do not satisfy the derivability criterion, can be pruned. That is, if g is not derivable from f in the corresponding rewrite system F, then such goals will never be satisfiable. This is expressed by the rule:

$$not-derivable(f,g): \{f(t_1,\ldots,t_n) \rightarrow^? g(s_1,\ldots,s_m)\} \cup E \Rightarrow FAIL$$

Thus the goal $rev(Y) \downarrow_? 1 \cdot nil$ (for which narrowing did not halt) can be pruned here after producing the one correct solution.

Putting all the above rules together, with some optimizations, we get the simple Prolog program for goal directed equation solving shown in Table 6.4 where *rite* is used for $\rightarrow^?$, and *unify* and *derivable* predicates are defined in the natural way. The first rule, which checks if the query term is a variable, is used to not allow restructuring in variables. By extending this idea, one could also capture basic narrowing by keeping track of the non-variable positions where restructurings are necessary.

There is still room for enhancements to the notion of operator rewriting, as can be seen from the following example:

$$\begin{array}{|rcl|} a(d(X)) & \rightarrow & b(X,X) \\ a(f(e)) & \rightarrow & f(e) \\ b(f(X),Y) & \rightarrow & b(X,f(Y)) \end{array}$$

Given the goal $a(f(U)) \downarrow_? b(V,e)$, narrowing and decomposition produce infinitely many (non-subsuming) equations when considering $b(V,e)$. Our notion of operator derivability can be used to detect that the only way for a term headed by a to join a term headed by b is for the first to reach the form $a(d(X))$, whereas there is no way for the subterm $f(U)$ of the left part of the goal to attain the form $d(X)$; hence, the goal is unsatisfiable.

6.4 Completeness of Equation Solving Procedures

Narrowing works by simulating each rewrite step in the solution $s\sigma \downarrow t\sigma$ by an application of a narrowing step. For irreducible solutions, no rewrite step takes place inside the σ and this procedure is complete for such solutions provided R is (ground) confluent and decreasing [Kaplan, 1987]. Without (ground) confluence, reducible solutions are lost. For example, if R is $\{f(a,b) \rightarrow c, a \rightarrow b\}$ or $\{f(x,g(x)) \rightarrow c, a \rightarrow g(a)\}$, then the goal $\{f(y,y) \downarrow_? c\}$ (which has a solution $\sigma = \{y \mapsto a\}$) cannot be transformed by either inference rule. For confluent and decreasing R, any most general unifier can be generated by keeping track of the substitutions introduced by narrowing.

This result easily extends to the decomposition and goal-directed procedures. Our simulation of innermost rewriting is complete for irreducible solutions. If R is a confluent and terminating conditional rewrite system, with no rule having a variable in the condition terms that is not present

in the left-hand side (a sufficient condition for this is decreasingness), then if $s\sigma \rightarrow t\sigma$ for some irreducible substitution σ, in this derivation all substitutions used for variables in conditions are also irreducible. Hence our procedure can simulate every rewrite step by a restructuring step.

For non-decreasing systems—even those that are terminating and confluent—new difficulties are introduced by the presence of "extra variables" in conditions as pointed out by Giovanetti and Moiso. Consider the following example which has an extra variable in a condition term.

$$
\begin{array}{rrcl}
 & a & \rightarrow & b \\
 & a & \rightarrow & c \\
f(X,b) = f(c,X) : & b & \rightarrow & c
\end{array}
$$

The third rule is feasible for an reducible substitution $\{X \mapsto a\}$ and so this system is indeed convergent as $b \rightarrow c$. Using any of the equation solving methods above we will not be able to solve a goal of the form $f(X,b) \downarrow_? f(c,c)$, (which does have an irreducible solution $\{X \mapsto c\}$) as we cannot prove that $b \rightarrow c$.

7 SUMMARY AND FUTURE WORK

We have studied conditional equational theories and how to use conditional rewriting to solve the validity problem and satisfiability problem. The main difficulty in directly lifting the results from unconditional systems to this setting is that it is no longer possible to distinguish easily between one-step of rewriting and a many-step derivation. This is because checking if a rule applies involves doing a proof that terms in the condition are equal.

By restricting our attention to decreasing systems, we can surmount this problem to some extent. For such systems, useful notions like rewriting are decidable, and confluence can be checked by just considering critical pairs as for unconditional systems. For non-decreasing systems, even without new variables in the condition terms, we showed that it is insufficient to just check the critical pairs. We need further restrictions on the rules, and we examined two such extensions–shallow-joinable and overlay systems.

The difficulties with checking confluence are also reflected in designing completion procedures to convert conditional equations to equivalent convergent rewrite systems. Even if we start with all the equations being decreasing, we often encounter non-decreasing critical pairs. We proposed a technique to handle such equations, by translating them to decreasing ones, using a conservative extension of the original theory.

For using conditional equations as a programming language it is very important to have efficient equation solving methods. We have identified some drawbacks with existing methods and suggested a goal-directed method that retains the top-down approach of the decomposition procedure (looking at subterms only when necessary), and incorporates oriented goals (to prevent narrowing non-query subterms) and pruning of some unsatisfiable goals—both in a uniform manner.

Several interesting problems still remain. Our confluence results seem close to optimal in the general setting. Practical methods for checking the confluence property need to be designed. Methods for testing and proving ground confluence will also be very useful. In equation solving, we would like to integrate eager rewriting and basic narrowing cleanly in our framework. The presence of extra variables in the conditions is crucial for capturing "logic" programming. But, in such systems, the equation solving methods that we have examined are not complete and need to be extended.

APPENDIX A

In this appendix, we describe a preliminary implementation of conditional completion and give a transcript of the example discussed in Chapter 5. For my Master's thesis, I designed and implemented RRL (a Rewrite Rule Laboratory) [Kapur and Sivakumar, 1984]. RRL is a system written in Franz/Zeta Lisp (compatible with both) and runs on VAX/SUN/Symbolics. Among its capabilities [Kapur and Zhang, 1989] are–

1. Completion Procedure to convert unconditional equations to equivalent rewrite systems.
2. Special completion techniques for theories with associative and commutative functions.
3. Verifying consitency and completeness of inductively defined structures and abstract data types.
4. Equational theorem proving methods for first order predicate calculus with equality.

I have now added to RRL the conditional completion procedure outlined in Chapter 5. Conditional equations are input in the form– $l \; == \; r \; if \; s_1 \; = \; t_1 \; \& \; \cdots \; \& s_n \; = \; t_n$ and unconditional ones as $l \; == \; r$. When RRL encounters a non-decreasing equation that it has to make into a rule, it uses the translation mechanism to pull equations from the condition to the left-hand side. It prompts the user to choose which equation from the condition to transfer to the left-hand side. Below is a transcript of a session on RRL running the $<$ example. There is some slight notational difference. Also, the contextual simplification we have implemented is not as powerful as described in Chapter 5.

```
aisune-11 %  rrl

        *******************************************
        ******  WELCOME TO REWRITE RULE LAB  *******
        *******************************************

Type Add, Akb, Auto, Clean, Complete, Delete, Dump, Franz, Init, Kb, Kb-option,
     List, Log, Norm, Operator, Order, Prove, Quit, Read, Reset, Rules, Stats,
     Synthesis, Meta, Undo, Unlog, Write or Help.
RRL-> ini

RRL is initialized.

Type Add, Akb, Auto, Clean, Complete, Delete, Dump, Franz, Init, Kb, Kb-option,
     List, Log, Norm, Operator, Order, Prove, Quit, Read, Reset, Rules, Stats,
     Synthesis, Meta, Undo, Unlog, Write or Help.
RRL-> ad
Type your equations in the format:  LHS == RHS  (eq. e * x == x)
                    or the format:  LHS == RHS if COND
Enter a ']' when done.
true & x == x
x & true == x
(x = x) == true
0 < s(0) == tt
s(x) < s(y) == x < y
x < z == tt if (((x < y) = tt) & ((y < z) = tt))
]

Equations succesfully read in were:
 1. (true & x) == x
 2. (x & true) == x
 3. (x = x) == true
 4. (0 < s(0)) == tt
 5. (s(x) < s(y)) == (x < y)
 6. (x < z) == tt  if  (((x < y) = tt) & ((y < z) = tt))

New constant set is: { true, tt, 0 }

Type Add, Akb, Auto, Clean, Complete, Delete, Dump, Franz, Init, Kb, Kb-option,
     List, Log, Norm, Operator, Order, Prove, Quit, Read, Reset, Rules, Stats,
     Synthesis , Meta, Undo, Unlog, Write or Help.
RRL-> oper pre < s 0 & = tt true

Precedence relation, < > s, is added.
Precedence relation, s > 0, is added.
```

```
Precedence relation, 0 > &, is added.
Precedence relation, & > =, is added.
Precedence relation, = > tt, is added.
Precedence relation, tt > true, is added.

Type Add, Akb, Auto, Clean, Complete, Delete, Dump, Franz, Init, Kb, Kb-option,
     List, Log, Norm, Operator, Order, Prove, Quit, Read, Reset, Rules, Stats,
     Synthesis, Meta, Undo, Unlog, Write or Help.
RRL->

 Trace mode now is -- Extended.

Type Add, Akb, Auto, Clean, Complete, Delete, Dump, Franz, Init, Kb, Kb-option,
     List, Log, Norm, Operator, Order, Prove, Quit, Read, Reset, Rules, Stats,
     Synthesis, Meta, Undo, Unlog, Write or Help.
RRL->
----- Step 1 -----

Processing  (true & x) == x

Adding Rule:
    [1] (true & x) ---> x

Processing  (x & true) == x

Adding Rule:
    [2] (x & true) ---> x

Processing  (x = x) == true

Adding Rule:
    [3] (x = x) ---> true

Processing  (0 < s(0)) == tt

Adding Rule:
    [4] (0 < s(0)) ---> tt

Processing  (s(x) < s(y)) == (x < y)

Adding Rule:
    [5] (s(x) < s(y)) ---> (x < y)

Processing  (x < z) == tt  if  (((x < y) = tt) & ((y < z) = tt))
```

```
In condition ((x < y) = tt) & ((y < z) = tt)

  which subterm do you wish to pick?
 --> 1

Adding Rule:
    [6] if((((x < y) = tt) & ((y < z) = tt)), (x < z))
                 ---> if((((x < y) = tt) & ((y < z) = tt)), tt)

Computing critical pairs with Rule:
     [6] (x < z) ---> tt  if  (((x < y) = tt) & ((y < z) = tt))

Computing critical pairs with Rule:
     [3] (x = x) ---> true

Computing critical pairs with Rule:
     [2] (x & true) ---> x

Computing critical pairs with Rule:
     [1] (true & x) ---> x

  Rules [2] and [1] yield following critical pair:
      true == true  if  (true & true)

Processing  true == true  if  (true & true)

Computing critical pairs with Rule:
     [4] (0 < s(0)) ---> tt

  Rules [4] and [6] yield following critical pair:
      (0 < z) == tt  if  (true & ((tt = tt) & ((s(0) < z) = tt)))

Processing  (0 < z) == tt  if  (true & ((tt = tt) & ((s(0) < z) = tt)))

Adding Rule:
    [7] if(((s(0) < z) = tt), (0 < z)) ---> if(((s(0) < z) = tt), tt)

Computing critical pairs with Rule:
     [7] (0 < z) ---> tt  if  ((s(0) < z) = tt)

Computing critical pairs with Rule:
     [5] (s(x) < s(y)) ---> (x < y)

  Rules [5] and [6] yield following critical pair:
      (s(x) < z) == tt  if  (true & (((x < y) = tt) & ((s(y) < z) = tt)))
```

```
  Rules [5] and [7] yield following critical pair:
      (0 < s(y)) == tt  if  (true & ((0 < y) = tt))

Processing  (0 < s(y)) == tt  if  (true & ((0 < y) = tt))

Adding Rule:
    [8] (0 < s(y)) ---> tt  if  ((0 < y) = tt)

Processing  (s(x) < z) == tt  if  (true & (((x < y) = tt) & ((s(y) < z) = tt)))

In condition ((x < y) = tt) & ((s(y) < z) = tt)

  which subterm do you wish to pick?
 --> 2

Adding Rule:
    [9] if((((x < y) = tt) & ((s(y) < z) = tt)), (s(x) < z))
                       ---> if((((x < y) = tt) & ((s(y) < z) = tt)), tt)

Computing critical pairs with Rule:
     [9] (s(x) < z) ---> tt  if  (((x < y) = tt) & ((s(y) < z) = tt))

  Rules [5] and [9] yield following critical pair:
      (s(x) < s(y)) == tt  if  (true & (((x < y1) = tt) & ((y1 < y) = tt)))

Processing  (s(x) < s(y)) == tt  if  (true & (((x < y1) = tt) & ((y1 < y) = tt)))

Computing critical pairs with Rule:
     [8] (0 < s(y)) ---> tt  if  ((0 < y) = tt)

  Rules [8] and [6] yield following critical pair:
      (0 < z) == tt  if  (((0 < y) = tt) & ((tt = tt) & ((s(y) < z) = tt)))

  Rules [4] and [8] yield following critical pair:
      tt == tt  if  (true & ((0 < 0) = tt))

Processing  tt == tt  if  (true & ((0 < 0) = tt))

Processing  (0 < z) == tt  if  (((0 < y) = tt) & ((tt = tt) & ((s(y) < z) = tt)))

In condition ((0 < y) = tt) & ((s(y) < z) = tt)

  which subterm do you wish to pick?
 --> 2
```

```
Adding Rule:
   [10] if((((0 < y) = tt) & ((s(y) < z) = tt)), (0 < z))
                     ---> if((((0 < y) = tt) & ((s(y) < z) = tt)), tt)

Computing critical pairs with Rule:
    [10] (0 < z) ---> tt  if  (((0 < y) = tt) & ((s(y) < z) = tt))

  Rules [5] and [10] yield following critical pair:
      (0 < s(y)) == tt  if  (true & (((0 < y1) = tt) & ((y1 < y) = tt)))

Processing  (0 < s(y)) == tt  if  (true & (((0 < y1) = tt) & ((y1 < y) = tt)))

Your system is canonical.

  [1] (true & x) ---> x
  [2] (x & true) ---> x
  [3] (x = x) ---> true
  [4] (0 < s(0)) ---> tt
  [5] (s(x) < s(y)) ---> (x < y)
  [6] if((((x < y) = tt) & ((y < z) = tt)), (x < z))
                         ---> if((((x < y) = tt) & ((y < z) = tt)), tt)
  [7] if(((s(0) < z) = tt), (0 < z)) ---> if(((s(0) < z) = tt), tt)
  [8] (0 < s(y)) ---> tt  if  ((0 < y) = tt)
  [9] if((((x < y) = tt) & ((s(y) < z) = tt)), (s(x) < z))
                         ---> if((((x < y) = tt) & ((s(y) < z) = tt)), tt)
 [10] if((((0 < y) = tt) & ((s(y) < z) = tt)), (0 < z))
                         ---> if((((0 < y) = tt) & ((s(y) < z) = tt)), tt)

Processor time used              = 1.25 sec
Number of rules generated        = 10
Number of critical pairs         = 7
Time spent in normalization      = 0.23 sec (18.67 percent of time)
Time spent while adding rules    = 0.20 sec (16.00 percent of time)
  (keeping rule set reduced)

Total processor time used (include 'undo' action) = 1.30 sec

Type Add, Akb, Auto, Clean, Complete, Delete, Dump, Franz, Init, Kb, Kb-option,
     List, Log, Norm, Operator, Order, Prove, Quit, Read, Reset, Rules, Stats,
     Synthesis, Meta, Undo, Unlog, Write or Help.
RRL-> q

Good bye siva.
```

APPENDIX B

In this appendix, we give a Prolog program that does goal-directed equation solving and show how it works on some examples. This is an interpreter for a conditional equational language like RITE [Dershowitz and Plaisted, 1988]. In addition to the features described in Chapter 6—oriented goals and pruning— this implementation also has eager rewriting and ignores non-normalised solutions that arise from restructuring subterms introduced from left-hand sides of rules by keeping track of basic positions. Also, we have implemented iterative depth first search to search the solution space fairly and enumerate all solutions.

```
/* This program does goal-directed equtaion solving.
   It also uses  eager rewriting and keeps track of basic positions.
   Top-level call is   rite(T,S) or rite(T,S,depth-bound)
   to find a sigma such that T sigma ->* S sigma (in depth-bound)
   If depth-bound is given we search only up to the bound, and will
   find all solutions within that depth.
   Otherwise, we do iterative deepening of DFS till we find all solutions.
*/

rite(X,Y) :-
         cputime(X1),
         basic(X,P) , isolve([[X,Y,P,0]],3),
         cputime(X2), X3 is X2 - X1, nl, write('Cpu: '),write(X3).

rite(X,Y,Bound) :-
          cputime(X1),basic(X,P) ,
          solve([[X,Y,P,0]],Bound),
          cputime(X2), X3 is X2 - X1, nl, write('Cpu: '),write(X3),nl.

/* does iterative DFS. keeps increasing bound till all solns found */
isolve(G, Bound) :-
          solve(G,Bound);
          (retract(bound)
                -> (B1 is Bound + 3,nl,
                   write('All solutions not exhausted.'), nl,
                   write('Increasing depth to '), write(B1),
                   write(' .. '),nl,isolve(G,B1))
                ; (nl,write('No more solutions possible. '),nl,fail)).
```

```
/* simplify all goals as much as possible wihtout backtracking, then do
   one step of either decomposition or restructuring of one of the goals*/
solve([],Bound) :- !.
solve(L,Bound) :-
           simp(L,L1), ! , onestep(L1,L2,Bound), solve(L2,Bound).

onestep([],[],Bound) :- !.
onestep(GL,L,Bound) :-
/*  pick a sub-goal  that was introduced at depth <= Bound */
             (pick(GL,G,Gr,Bound) ->
               addsubgoals(G,Gr,L);
               ((bound -> true; assert(bound)),fail)).

addsubgoals(G,Gr,Gl) :- decomp(G,Gr,Gl,1) ; restruct(G,Gr,Gl) .

/* decompose goal-- same top operators */
decomp(G,Gr,Gl,Flag) :-
          G = [X, Y, Xb,D], (var(Flag) -> D1 is D ; D1 is D + 1),
          functor(X,F,N), functor(Y,F,N),
          X =.. [F | Ax], Xb =.. [F | Axb],
          Y =.. [F | Ay], addgoals(D1,Ax,Ay,Axb,Gr,Gl).

/* restructure -- apply a rule at the top-operator */
restruct(G,Gr,Gl) :-
          G = [X, Y, Xb,D], D1 is D + 1,
          functor(X,F,N), functor(L,F,N), getrule(L,R,C),
          X =.. [F | Ax], Xb =.. [F | Axb], L =.. [F | Al],
          basic(R,Rp), basic(C,Cp),
          addgoals(D1,[C|Ax],[true|Al],[Cp|Axb],[[R,Y,Rp,D1]|Gr],Gl).


/* simp(Glist1,Glist2) - Glist2 is the simplified form of Glist1 */
simp([],[]) :- !.
simp([G | Gr] , L) :-
         (forced(G,X,Y) -> (dunify(X,Y),simp(Gr,L)) ;
           (possible(G) ->
             (decomposable(G,Gr,Gl) -> simp(Gl,L) ;
                         (simp(Gr,Gl), L = [G | Gl])))).

/* have omitted eager rewriting to check speed */
/* simplify(Goal,GList)
   -- we do pruning, forced decompositions, eager rewriting */
simplify(G, []) :- forced(G,X,Y), !, dunify(X,Y).
simplify(G,Glist) :- (decomposable(G,Gl) -> loop2(simplify,Gl,Gl1),
```

```
                                        mergeall(Gl1,Glist), ! ;
                      bnorm(G,G1),!,
                      (G == G1 -> Glist = [G1] ; simplify(G1,Glist))).

/* top-level operators are ok */
possible(G) :-   G = [X, Y, Xb, N], possible(X,Y).

forced(G,X,Y) :- G = [X, Y, e, N]. /* have reached basic position */

/* reachable(F1,N1,F2) -- A term with top operator F1 (of arity N1) can be
  converted to a term with top operator F2. Used for pruning goals */
reachable(F,N,F) :- ! .
reachable(F1,N1,F2) :-
                   functor(L,F1,N1), getrule(L,R,C),
                   (var(R) -> ! ; (functor(R,F,N), reachable(F,N,F2))).

/* possible(X,Y) - Y's top-operator is reachable from
                   X's top-operator */
possible(X,Y) :- (var(X); var(Y)) -> true ;
                 (functor(X,F1,N1),functor(Y,F2,N2),reachable(F1,N1,F2)).


decomposable(G,Gr,Gl) :-
         G = [X, Y, Xb, D], functor(X,F,N),
         functor(L,F,N),not(getrule(L,R,C)),decomp(G,Gr,Gl,F1).

addgoals(N, [],[],[],G,G) :- !.
addgoals(N, [X|Xr], [Y | Yr], [Z | Zr], G, Go) :-
                      (X == Y -> addgoals(N,Xr,Yr,Zr,G,Go);
                        (Z == e -> (dunify(X,Y), addgoals(N,Xr,Yr,Zr,G,Go));
                          addgoals(N,Xr,Yr,Zr,[[X,Y,Z,N] | G],Go))).


/* pick a subgoal introduced at depth <= Bound */
pick([G | Gr], G1, Gl,Bound) :-
            G = [X,Y,Z,N],
            (N =< Bound
              -> (G1 = G , Gl = Gr) ;
              (pick(Gr,G1,Gl1,Bound), Gl = [G | Gl1])).

bnorm([X,Y,Xb,N],[Z,Y,Zb,N]) :-  bnorm(X,Xb,Z,Zb).

/* some utilities like unification, matching and eager rewriting
are defined below */
copy(Term,Copy) :- assert($(Term)), retract($(Copy)).
```

```
loop2(P,[],[]) :- !.
loop2(P,[T | Tr],[S | Sr]) :- X =.. [P | [T,S]],call(X), loop2(P,Tr,Sr).

loop3(P,[],[],[]) :- !.
loop3(P,[T|Tr],[B|Br],[S|Sr]) :- X =.. [P|[T,B,S]], call(X),loop3(P,Tr,Br,Sr).

loop4(P,[],[],[],[]) :- !.
loop4(P,[T|Tr],[B|Br],[S|Sr],[C|Cr]) :-
              X =.. [P|[T,B,S,C]], call(X),loop4(P,Tr,Br,Sr,Cr).

/* basic(X,Y) - Y is the basic (non-var) positions in X. e is for vars */
basic(X,Y) :-
          (var(X) -> Y = e ;
            (atomic(X) -> Y = X ;
              (X =.. [F | AX], loop2(basic,AX,AY), Y =.. [F | AY]))).

occurs(Var,Term) :-
        (Var == Term -> true ;
           (var(Term) -> fail ;
              (Term =.. [F | AS] , occurs-any(Var,AS)))).

occurs-any(Var,[]) :- fail .
occurs-any(Var,[T|Trest]) :-
        (occurs(Var,T) -> true; occurs-any(Var,Trest)) .


/* assumes vars in pattern do not appear in target */
dmatch(X,Y) :- match(X,Y,[],Binds), dobind(Binds), !.

match(X,Y,InB,OB) :- (X == Y -> OB = InB ;
                       (var(X) -> addbind(X,Y,InB,OB) ;
                         (var(Y) -> fail ;
                           (X =.. [F | AX] , Y =.. [F | AY],
                           allmatch(AX,AY,InB,OB))))).

allmatch([],[],InB,InB).
allmatch([X1 | Xr],[Y1|Yr],I,O) :- match(X1,Y1,I,O1),!,
                                   allmatch(Xr,Yr,O1,O).

dobind([]).
dobind([[X | Y] | Z]) :- X = Y, dobind(Z).

addbind(X,Y,[],[[X | Y]]).
addbind(X,Y,[[U | V] | Z],[[U | V] | Z1]) :-
```

```
                             (X == U -> Y == V, Z = Z1 ;
                                addbind(X,Y,Z,Z1)).

dunify(X,Y) :- (X == Y -> true ;
                 (var(X) -> not(occurs(X,Y)),  X = Y ;
                   (var(Y) -> not(occurs(Y,X)) , Y = X ;
                     X =.. [F | AX] , Y =.. [F | AY], loop2(dunify,AX,AY)))).

norm(X,Y) :- basic(X,Xb), inorm(X,Xb,Y).

inorm(X,e,X) :-  !.
inorm(X,Xb,Y) :- X =.. [F | AX], Xb =.. [F | Ab],
               loop3(inorm,AX,Ab,AY), Y1 =.. [F | AY],
               (rewrites-top(Y1,Y2,Yb) -> inorm(Y2,Yb,Y) ; Y = Y1).
rewrites-top(Y1,Y2,Yb) :- functor(Y1,F,N), functor(L,F,N),
                   getrule(L,Y2,C), basic(C,Cb), basic(Y2,Yb),
                   dmatch(L,Y1), inorm(C,Cb,true),!.

bnorm(X,e,X,e) :- ! .
bnorm(X,Xb,Y,Yb) :- X =.. [F | Ax], Xb =.. [F | Axb],
                    loop4(bnorm,Ax,Axb,Ay,Ayb),
                    Y1 =.. [F | Ay], Y1b =.. [F | Ayb],
                    (rewrites-top(Y1,Y1b,Y2,Y2b) ->
                          bnorm(Y2,Y2b,Y,Yb) ;
                          Y = Y1, Yb = Y1b).

rewrites-top(Y,Yb,Y1,Y1b) :-
       functor(Y,F,N), functor(L,F,N), getrule(L,R,C),
       match(L,Y,[],Binds),!,copy(L,R,Lc,Rc), basic(C,Cb),
       dobind(Binds), inorm(C,Cb,true), !, Y1 = R,
       bmatch(Lc,Yb), Y1b = Rc .

copy(L,R,Lc,Rc) :- T =.. [a | [L, R]], copy(T,T1),
                   T1 =.. [a | [Lc, Rc]].

bmatch(X,e) :- ! .
bmatch(X,Y) :- var(X), !, X = Y .
bmatch(X,Y) :- X =.. [F | AX], Y =.. [F | AY], loop2(bmatch,AX,AY).

mergeall([],[]) :- ! .
mergeall([L | Lr], Lr1 ) :- mergeall(Lr,Lr2), append(L,Lr2,Lr1).

append([],X,X).
append([A | X],Y,[A | Z]) :- append(X,Y,Z).
```

```
/* rules */
/* Rules are written of the form  rule(L,R) if unconditional or
   rule(L,R,C) if conditional */

getrule(L,R,true) :- rule(L,R).
getrule(L,R,C) :- rule(L,R,C).

/* conditional rules that define quotient and mod for natural numbers */
rule(quot(X,Y),s(quot(diff(X,Y),Y)),ge(X,Y)).
rule(quot(X,Y),0,gt(Y,X)).

rule(mod(X,Y),X, not(ge(X,Y))).
rule(mod(X,X),0).
rule(mod(X,Y),mod(diff(X,Y),Y),and(gt(Y,0),not(ge(Y,X)))).

rule(diff(X,0),  X).
rule(diff(s(X),s(Y)),  diff(X,Y)).

rule(add(X,0),X).
rule(add(X,s(Y)),s(add(X,Y))).

rule(ge(X,0),true).
rule(ge(0,s(Y)),false).
rule(ge(s(X),s(Y)),ge(X,Y)).

rule(lt(X,Y),not(ge(X,Y))).
rule(le(X,Y),ge(Y,X)).
rule(gt(X,Y),not(ge(Y,X))).

rule(ap(nil,X), X).
rule(ap(+(X,Y),Z), +(X,ap(Y,Z))).

rule(f(f(X)), f(X)).
rule(g(f(X),Y), g(d,Y)).
rule(g1(f1(a)), a).


rule(not(true),false).
rule(not(false),true).
rule(eq(X,X),true).
rule(and(true,X),X).
```

Below is a transcript of a session that uses this equation solver for some simple examples.

```
aisune-19 % sbprolog
SB-Prolog Version 2.3.1
```

```
| ?- ['gdir.pr'].
yes
| ?- rite(ap(X,Y),+(1,nil)).

Cpu: 300
X = nil
Y = 1 + nil;

Cpu: 660
X = 1 + nil
Y = nil;

No more solutions possible.
no
| ?- rite(ap(X,ap(Y,Z)),+(1,+(2,+(3,nil)))).

Cpu: 280
X = nil
Y = nil
Z = 1 + (2 + (3 + nil));

Cpu: 560
X = nil
Y = 1 + nil
Z = 2 + (3 + nil);

Cpu: 820
X = nil
Y = 1 + (2 + nil)
Z = 3 + nil;

Cpu: 2220
X = 1 + nil
Y = nil
Z = 2 + (3 + nil);

Cpu: 2520
X = 1 + (2 + nil)
Y = nil
Z = 3 + nil;

Cpu: 2760
X = 1 + (2 + (3 + nil))
Y = nil
Z = nil;
```

```
Cpu: 4060
X = 1 + nil
Y = 2 + nil
Z = 3 + nil;

Cpu: 4360
X = 1 + (2 + nil)
Y = 3 + nil
Z = nil;

Cpu: 6040
X = 1 + nil
Y = 2 + (3 + nil)
Z = nil;

All solutions not exhausted.
Increasing depth to 6 ..

Cpu: 8000
X = nil
Y = nil
Z = 1 + (2 + (3 + nil));

Cpu: 8320
X = nil
Y = 1 + nil
Z = 2 + (3 + nil);

Cpu: 8600
X = nil
Y = 1 + (2 + nil)
Z = 3 + nil;

Cpu: 8860
X = nil
Y = 1 + (2 + (3 + nil))
Z = nil;

Cpu: 10040
X = 1 + nil
Y = nil
Z = 2 + (3 + nil);

Cpu: 10340
```

```
X = 1 + (2 + nil)
Y = nil
Z = 3 + nil;

Cpu: 10580
X = 1 + (2 + (3 + nil))
Y = nil
Z = nil;

Cpu: 11860
X = 1 + nil
Y = 2 + nil
Z = 3 + nil;

Cpu: 12120
X = 1 + (2 + nil)
Y = 3 + nil
Z = nil;

Cpu: 13780
X = 1 + nil
Y = 2 + (3 + nil)
Z = nil;

No more solutions possible.
no

/* Below is an example that uses  conditional rules.
We begin to enumerate all positive odd numbers
by asking for X such that X mod 2 is 1  */

| ?- rite(mod(X,s(s(0))), s(0)).

Cpu: 860
X = s(0);

All solutions not exhausted.
Increasing depth to 6 ..

Cpu: 15360
X = s(0);

Cpu: 18360
X = s(s(s(0)));
```

```
Cpu: 23680
X = s(s(s(s(s(0)))));

All solutions not exhausted.
Increasing depth to 9 ..

Cpu: 47700
X = s(0);

Cpu: 50660
X = s(s(s(0)));

Cpu: 55960
X = s(s(s(s(s(0)))));

Cpu: 61260
X = s(s(s(s(s(s(s(0)))))))
yes
```

REFERENCES

[Bachmair, 1987] Bachmair, L. "Proof methods for equational theories," *Ph.D. thesis*, Department of Computer Science, University of Illinois Urbana, IL, 1987.

[Bachmair, *et al.*, 1986] Bachmair, L., Dershowitz, N. and Hsiang J., "Orderings for equational proofs," *Proceedings of the Symposium on Logic in Computer Science*, Cambridge, MA, pp. 346-357, June 1986.

[Bachmair, *et al.*, 1989] Bachmair, L., Dershowitz, N. and Plaisted, D. A. "Completion without failure," *Resolution of Equations in Algebraic Structures*, H. Ait-Kaci and M. Nivat, editors, Academic Press, New York, pp. 1-30, 1989.

[Bergstra and Klop, 1982] Bergstra, J. A., and Klop, J. W. "Conditional rewrite rules: Confluence and termination," Report IW 198/82 MEI, Mathematische Centrum, Amsterdam, 1982.

[Bergstra and Klop, 1986] Bergstra, J. A., and Klop, J. W. "Conditional rewrite rules: confluency and termination," *J. of Computer and System Sciences* **32** , pp. 323-362 , 1986.

[Bosco *et al.*, 1987] Bosco, P. G., Giovanetti, E., and Moiso, C. "Refined strategies for semantic unification," *Proceedings of International Joint Conference on Theory and Practice of Software Development*, Pisa, Italy (March 1987), pp. 276-290. (Available as Vol. 250, Lecture Notes in Computer Science, Springer, Berlin.)

[Brand, *et al.*, 1978] Brand, D., Darringer, J. A., and Joyner, W. J. "Completeness of conditional reductions," Report RC 7404, IBM Thomas J. Watson Research Center, December 1978.

[Dershowitz, 1984] Dershowitz, N. "Equations as programming language," *Proceedings of the Fourth Jerusalem Conference on Information Technology* Computer Society, pp. 114-123, May 1984.

[Dershowitz, 1985] Dershowitz, N. "Computing with rewrite systems," *Information and Control*, **64** (2/3), pp. 122-157, May/June 1985.

[Dershowitz, 1987] Dershowitz, N., "Termination of rewriting," *J. of Symbolic Computation*, **3** (1 / 2), pp. 69-115, February/April 1987. (Corrigendum [December 1987], Vol. 4, No. 3, pp. 409-410.)

[Dershowitz, 1989] Dershowitz, N. "Completion and its applications," *Resolution of Equations in Algebraic Structures*, Academic Press, New York, pp. 31-86, 1989.

[Dershowitz and Jouannaud, 1989] Dershowitz, N., and Jouannaud, J.-P. "Rewrite Systems," *Handbook of Theoretical Computer Science,* North-Holland, 1989. (To appear.)

[Dershowitz and Okada, 1988] Dershowitz, N., and Okada, M. "Proof-theoretic techniques and the theory of rewriting," *Proc. of the Third Symposium on Logic in Computer Science,* Edinburgh, Scotland, pp. 104-111, July 1988.

[Dershowitz and Okada, 1988b] Dershowitz, N., and Okada, M. "Conditional equational programming and the theory of conditional term rewriting," *Proc. of the International Conference on Fifth Generation Computer Systems,* Tokyo, Japan, November 1988.

[Dershowitz and Plaisted, 1985] Dershowitz, N., and Plaisted, D. A. "Logic programming cum Applicative Programming," *Proceedings of the 1985 Symposium on Logic Programming*, Boston, MA. pp. 54-66, July 1985.

[Dershowitz and Plaisted, 1988] Dershowitz, N., and Plaisted, D. A. "Equational programming," In: *Machine Intelligence 11* (J. E. Hayes, D. Michie, and J. Richards, eds.), Oxford Press, Oxford, pp. 21-56, 1988.

[Dershowitz and Sivakumar, 1987] Dershowitz, N., and Sivakumar, G. "Solving goals in equational languages," *First International Workshop on Conditional Rewriting Systems*, Orsay, France, pp. 45-55, July 1987. (Available as Vol. 308, *Lecture Notes in Computer Science*, Springer, Berlin.)

[Dershowitz and Sivakumar, 1987] Dershowitz, N., and Sivakumar, G. "Goal-directed Equation Solving," *Proceedings of the Seventh National Conference on Artificial Intelligence,* St. Paul, MN, pp. 166-170, August 1988.

[Dershowitz, *et al.*, 1987] Dershowitz, N., Okada, M., and Sivakumar, G. "Confluence of Conditional Rewrite Systems," *First International Workshop on Conditional Rewriting Systems*, Orsay, France, pp. 31-44, July 1987. (Available as Vol. 308, *Lecture Notes in Computer Science*, Springer, Berlin.)

[Dershowitz, *et al.*, 1988] Dershowitz, N., Okada, M., and Sivakumar, G. "Canonical conditional rewrite systems," *Proceedings of the Ninth Conference on Automated Deduction*, Argonne, IL, pp. 538-549, May 1988 (Available as Vol. 310, *Lecture Notes in Computer Science*, Springer, Berlin.)

[Fay, 1979] Fay, M. "First-order unification in an equational theory," *Proceedings of the Fourth Workshop on Automated Deduction*, Austin, TX (February 1979), pp. 161-167.

[Fribourg, 1985] Fribourg, L. "SLOG: A logic programming language interpreter based on clausal superposition and rewriting," *Proceedings of the 1985 Symposium on Logic Programming*, Boston, MA (July 1985), pp. 172-184.

[Ganzinger, 1987] Ganzinger, H., "A Completion Procedure for Conditional Equations," *First International Workshop on Conditional Rewriting Systems*, Orsay, France, pp. 62-83, July 1987. (Available as Vol. 308, Lecture Notes in Computer Science, Springer, Berlin.)

[Goguen and Meseguer, 1986] Goguen, J. A., and Meseguer, J. "EQLOG: Equality, types and generic modules for logic programming," In *Logic Programming: Functions, relations and equations* (D. DeGroot and G. Lindstrom, eds.), Prentice-Hall, Englewood Cliffs, NJ, pp. 295-363, 1986.

[Hsiang, 1982] Hsiang, J., "Topics in automated theorem proving and program generation," Ph. D. thesis, Report R-82-1113, Department of Computer Science, University of Illinois, Urbana, IL, December 1982.

[Hsiang and Rusinowitch, 1987] Hsiang, J., and Rusinowitch, M., "On word problems in equational theories," *Proceedings of the Fourteenth EATCS International Conference on Automata, Languages and Programming*, Karlsruhe, West Germany, pp. 54-71, July 1987.

[Huet, 1981] Huet, G., "A complete proof of correctness of the Knuth-Bendix completion algorithm," *J. Computer and Systems Sciences*, Vol. 23, No. 1, pp. 11-21, 1981.

[Huet and Oppen, 1980] Huet, G., and Oppen, D. C., "Equations and rewrite rules: A survey," *Formal Language Theory: Perspectives and Open Problems*, ed. R. Book, Academic Press, New York, 1980, pp. 349-405.

[Hullot, 1980] Hullot, J. M. "Canonical forms and unification," *Proceedings of the Fifth Conference on Automated Deduction*, Les Arcs, France (July 1980), pp. 318-334.

[Jouannaud and Waldmann, 1986] Jouannaud, J. P., and Waldmann, B. "Reductive Conditional term rewriting systems," *Proceedings of the Third IFIP Working Conference on Formal Description of Programming Concepts*, Ebberup, Denmark.

[Kaplan, 1984] Kaplan, S. "Fair conditional term rewriting systems: Unification, termination and confluency," Laboratoire de Recherche en Informatique, Université de Paris-Sud, Orsay, France, November 1984.

[Kaplan, 1987] Kaplan, S. "Simplifying conditional term rewriting systems: Unification, termination and confluence," *Journal of Symbolic Computation*, 1987 4(3), pp. 295-334.

[Kaplan and Rémy, 1989] Kaplan, S. and Rémy, J. L., "Completion algorithms for conditional rewriting systems," *Resolution of Equations in Algebraic Structures*, H. Ait-Kaci and M. Nivat, editors, Academic Press, New York, 1989.

[Kapur and Sivakumar, 1984] Kapur, D., and Sivakumar, G. "Experiments with and architecture of RRL, a rewrite rule laboratory," *Proceedings of an NSF Workshop on the Rewrite Rule Laboratory*, Schenectady, NY (September 1983), pp. 33-56. (Available as Report 84GEN008, General Electric Research and Development [April 1984].)

[Kapur and Zhang, 1986] Kapur, D. and Zhang, H., "An overview of Rewrite Rule Laboratory (RRL)," *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, Chapel Hill, NC, pp. 559-563, April 1989. (Available as Vol. 355, Lecture Notes in Computer Science, Springer, Berlin.)

[Kirchner, 1985] Kirchner, C., "Methodes et outils de conception systematique d'algorithmes d'unification dans les theories equationnelles," *These d'Etat,*, Université de Nancy, June 1985.

[Klop, 1987] Klop, J. W., "Term rewriting systems: A tutorial," *J Bulletin of the European Association for Theoretical Computer Science*, June 1987, Vol. 32, pp. 143-183.

[Knuth and Bendix, 1970] Knuth, D. E., and Bendix, P. B. "Simple word problems in universal algebras," In: *Computational Problems in Abstract Algebra*, J. Leech, ed. Pergamon Press, Oxford, U. K., 1970, pp. 263-297.

[Kounalis and Rusinowitch, 1988] Kounalis, E., and Rusinowitch, M., "On word problems in Horn theories," *Proceedings of the Ninth Conference on Automated Deduction*, Argonne, IL, pp. 526-537, May 1988 (Available as Vol. 310, *Lecture Notes in Computer Science*, Springer, Berlin.)

[Lankford, 1975] Lankford, D. S., "Canonical Inference," Memo ATP-25, Automatic Theorem Proving Project, University of Texas, Austin, TX, May 1975.

[Martelli, *et al.*, 1986] Martelli, A., Moiso, C. and Rossi, G. F. "An algorithm for unification in Equational Theories," *Proceedings of the Third IEEE Symposium on Logic Programming*, Salt Lake City, UT (September 1986), pp. 180-186.

[Musser, 1980] Musser, D. R., "On proving inductive properties of abstract data types," *Proceedings of the Seventh ACM Symposium on Principles of Programming Languages*, 1980, Las Vegas, NV, pp. 154-162.

[Newman, 1942] Newman, M. H. A., "On theories with a combinatorial definition of equivalence," *Annals of Mathematics* **43** (2), pp. 223-243, 1942.

[O'Donnell, 1985] O'Donnell, M. J., "Equational logic as a programming language," MIT Press, Cambridge, Mass., 1985.

[Okada, 1987] Okada, M. "A simple relationship between Buchholz's new system of ordinal notations and Takeuti's system of ordinal diagrams," *Journal of Symbolic Logic* 52 (1987).

[Plaisted, 1987] Plaisted, D. A., "A Logic for Conditional Term Rewriting Systems," *First International Workshop on Conditional Rewriting Systems*, Orsay, France (July 1987), pp. 212-227. (Available as Vol. 308, Lecture Notes in Computer Science, Springer, Berlin.)

[Reddy, 1986] Reddy, U. S., "On the relationship between logic and functional languages," *Logic Programming: Functions, Relations, and Equations*, ed. D. DeGroot and G. Lindstrom, Prentice-Hall, Englewood Cliffs, NJ, pp. 3-36, 1986.

[Rémy, 1982] Rémy J.-L., "Etude des systèmes de reéécriture conditionnels et applications aux types abstraits algébriques," Thèse, Institut National Polytechnique de Lorraine, July 1982.

[Réty, 1987] Réty, P. "Improving basic narrowing techniques," *Proceedings of the Second International Conference on Rewriting Techniques and Applications*, Bordeaux, France (May 1987), pp. 228-241. (Available as Vol. 256, Lecture Notes in Computer Science, Springer, Berlin.)

[Réty, *et al.*, 1985] Réty, P., Kirchner, C., Kirchner, H., and Lescanne, P. "NARROWER: A new algorithm for unification and its application to logic programming," *Proceedings of the First International Conference on Rewriting Techniques and Applications*, Dijon, France (May 1985), pp. 141-157. (Available as Vol. 202, Lecture Notes in Computer Science, Springer, Berlin [September 1985].)

[Slagle, 1974] Slagle, J. R., "Automated theorem-proving for theories with simplifiers, commutativity, and associativity," *J. of the Association for Computing Machinery,* Vol. 21, No. 4, pp. 622-642, 1974.

[Toyama, 1987] Toyama, Y. "Term Rewriting Systems with membership conditions," *First International Workshop on Conditional Rewriting Systems*, Orsay, France, July 1987. (Available as Vol. 308, Lecture Notes in Computer Science, Springer, Berlin.)

[Zhang and Rémy, 1985] Zhang, H., and Rémy, J. L. "Contextual rewriting," *Proceedings First International Conference on Rewriting Techniques and Applications*, Dijon, France (May 1985), pp. 46-62. (Available as Vol. 202, Lecture Notes in Computer Science, Springer, Berlin [September 1985].)

VITA

G. Sivakumar was born on September 30, 1960 in Madurai, India. He received his Bachelor of Technology degree in Electrical Engineering, in 1984, from the Indian Institute of Technology, Madras, and the Master of Science degree in Computer Science, in 1984, from Renssealaer Polytechnic Institute, New York. He then joined the University of Illinois at Urbana-Champaign for his doctoral studies. On completing his Ph.D., he joined the the faculty of the Department of Computer and Information Science, at the University of Delaware.