SEMANTIC UNIFICATION FOR CONVERGENT SYSTEMS

BY

SUBRATA MITRA

B.Tech., Indian Institute of Technology, Kanpur, 1988
M.S., University of Delaware, 1991

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, 1994

Urbana, Illinois

SEMANTIC UNIFICATION FOR CONVERGENT SYSTEMS

Subrata Mitra, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1994
Nachum Dershowitz, Advisor

Equation solving is the process of finding a substitution of terms for variables that makes two terms equal in a given theory, while *semantic unification* is the process that generates a basis set of such unifying substitutions. A simpler variant of the problem is *semantic matching*, where the substitution is made in only one of the terms. Semantic unification and matching constitute an important component of theorem proving and programming language interpreters.

In this thesis we formulate a unification procedure based on a system of transformation rules that looks at goals in a *lazy*, *top-down* fashion, and prove its soundness and completeness for equational theories described by convergent rewrite systems (finite sets of equations that compute unique output values when applied from left-to-right to input values).

We consider different variants of the system of transformation rules. We describe syntactic restrictions on the equations under which simpler sets of transformation rules are sufficient for generating a complete set of semantic matchings. We show that our first-order unification procedure, with slight modifications, can be used to solve the satisfiability problem in combinatory logic together with a convergent set of algebraic axioms, resulting in a complete higher-order unification procedure for the given algebra. We also provide transformation rules to handle situations where some of the function symbols additionally satisfy the equivalences of associativity and commutativity.

Termination of a system of directed equations is essential for proving existence and uniqueness of normal forms. Furthermore, termination is essential for simplification in theorem provers. We provide a simple restriction on the well-known "recursive path ordering" which can be used for proving termination of extended rewriting, modulo the axioms of associativity and commutativity.

Finally, we formulate various syntactic and semantic conditions on the given equations and the goal which result in decidability of semantic matching. We also investigate decidable cases of semantic unification.

ACKNOWLEDGEMENTS

I would like to thank Prof. G. Sivakumar and Prof. Nachum Dershowitz for getting me interested in the general area of term rewriting, and also for their encouragement and support. The presentation of this work has been greatly improved by numerous suggestions made by Prof. Nachum Dershowitz.

I am also indebted to Hubert Comon, Deepak Kapur, Claude Kirchner and a number of their students for their suggestions on different drafts of papers which previously reported results contained herein.

The research reported in this thesis was supported in part by the U. S. National Science Foundation, under Grants CCR-90-07195 and CCR-90-24271.

TABLE OF CONTENTS

LIST OF TABLES

1 INTRODUCTION

1.1 Motivation

Functional programming (see [Henderson, 1980] for an introductory exposition), which originated from the study of the Lambda-Calculus, is a style of programming based on recursive function definitions. For example, in a typical functional programming language such as ML (see [Paulson, 1991] for the syntax), we could define *append* and *reverse* on lists as:

$$
\begin{array}{l}
\textbf{val rec } \mathit{append} = \\
\quad \textbf{fn}(x, y) \Rightarrow \\
\qquad \textbf{if } x = \mathit{nil} \textbf{ then } y \textbf{ else } \mathit{hd}\, x \cdot \mathit{append}(\mathit{tl}\, x, y), \\
\textbf{val rec } \mathit{reverse} = \\
\quad \textbf{fn}(x) \Rightarrow \\
\qquad \textbf{if } x = \mathit{nil} \textbf{ then } \mathit{nil} \textbf{ else } \mathit{append}(\mathit{reverse}(\mathit{tl}\, x), \mathit{hd}\, x \cdot \mathit{nil}).
\end{array}
$$

(with the operations *hd* and *tl* for selecting the first element and the remainder of the list, respectively, *nil* for the empty list, and $\cdot$ for list constructing). Evaluating an expression like $\mathit{reverse}(1 \cdot 2 \cdot \mathit{nil})$ results in the value $2 \cdot 1 \cdot \mathit{nil}$.

Given the definitions of *append* and *reverse*, it is natural to define predicates for checking if a list represents a palindrome:

$$
\begin{array}{rcl}
\mathit{palindrome}(\mathit{append}(x, \mathit{reverse}(x))) & = & \mathit{true}, \\
\mathit{palindrome}(\mathit{append}(x, a \cdot \mathit{reverse}(x))) & = & \mathit{true}.
\end{array}
$$

We get these definitions for "free" once we have those for *append* and *reverse*. However, in order to use such definitions in a functional language, it is necessary to match patterns of the form $\mathit{append}(x, \mathit{reverse}(x))$ to values like $1 \cdot 2 \cdot 2 \cdot 1 \cdot \mathit{nil}$. To perform such matchings (which is not possible in current functional languages), a decidable matching algorithm is required.

Functional programming is an elegant alternative to the more traditional imperative style because of its declarative nature. It does not involve the notions of "states" and "side-effects," which are common to more traditional programming. Other advantages of this paradigm in-

clude implicit parallelism (different arguments of a function may be evaluated independently), the ability to deal with function-valued arguments and outputs (higher-order capability) and simplicity (due to the complete lack of control structures).

Logic programming is another programming methodology with similar features, which grew as an outcome of efforts to mechanize mathematical logic. See [Kowalski, 1979] for a survey on logic programming. Logic programs, which use Horn clauses to recursively define predicates, are elegant in their use of "logic-variables." For example, we could define *append* in PROLOG (see [Clocksin and Mellish, 1981]) as:

$$\begin{aligned} append(nil, x, x) &\text{ :- } true, \\ append(a \cdot x, y, a \cdot z) &\text{ :- } append(x, y, z) \end{aligned}$$

and solve the query $append(x, x, x) \stackrel{?}{=} true$ to obtain the solution $\{x \mapsto nil\}$.

Although functional and logic programming both fall in the category of declarative languages, there are some significant differences: The presence of logic-variables in the latter makes it more expressive, whereas functional programming entails higher-order capabilities and has simpler operational semantics. It is, therefore, of interest to combine the essential features of these two programming paradigms. Over the years, numerous different approaches have been suggested for this combination, including [Barbuti *et al.*, 1986; Bellia and Levi, 1986; Reddy, 1986; Subrahmanyam and You, 1986]; see [DeGroot and Lindstrom, 1986] for a survey of the area. One popular idea is based on conditional equational theories [Dershowitz and Josephson, 1984; Goguen and Meseguer, 1984; Fribourg, 1985; Lindstrom, 1985; Dershowitz and Plaisted, 1988; Cheong and Fribourg, 1993]. In the equational programming paradigm proposed by Dershowitz and Plaisted [1988] (see also [Josephson and Dershowitz, 1989; Dershowitz and Okada, 1990]), computation consists of solving an equation in the theory specified by the program (program being a set of conditional equations that define unique normal forms). For example, consider the definition of addition ($+$) over natural numbers (natural numbers being represented in unary notation, using the constant 0 and the successor function s):

$$\begin{aligned} 0 + x &= x, \\ s(x) + y &= s(x + y). \end{aligned}$$

Given this system of equations, a query of the form $x \stackrel{?}{=} s(s(0)) + (s(0) + s(0))$ can be solved (to give the solution $\{x \mapsto s(s(s(s(0))))\}$) by evaluating the right-hand side of the goal by using the equations as left-to-right rewrite rules. (We are only interested in solutions that do not involve $+$.) This type of a query corresponds to functional-programming, where the inputs are fully instantiated, and *rewriting* (based on the notion of "replacements of equals by equals") is the evaluation mechanism. On the other hand, for a query like $x + x \stackrel{?}{=} y + s(s(0))$, we have to use an equation solver to find the solutions $\{x \mapsto s(0), y \mapsto 0\}$, $\{x \mapsto s(s(0)), y \mapsto s(s(0))\}$, and so on. This corresponds to the logic-programming capability of a PROLOG-like language. Furthermore, given the definition of addition we could define subtraction $(-)$, using a conditional equation, as

$$x = y + z \quad : \quad x - y \quad = \quad z.$$

We could now evaluate $s(s(s(0))) - s(0) \stackrel{?}{=} z$ by solving the matching goal $s(s(s(0))) \stackrel{?}{=} s(0) + z$ to get the solution $\{z \mapsto s(s(0))\}$. Notice that the definitions of *append* and *reverse* can also be formulated as (unconditional) equations

$$append(nil, y) = y,\ append(a \cdot x, y) = a \cdot append(x, y),$$

$$reverse(nil) = nil,\ reverse(a \cdot x) = append(reverse(x), a \cdot nil).$$

Equation solving is the process of finding a substitution that makes two terms equal in a given theory, while *semantic unification* is the process that generates a *basis* set (for any solution to the goal, the basis set must subsume one that is equivalent in the underlying theory) of such unifying substitutions. Here, we formulate a unification procedure, based on transformation rules, that looks at terms in a *lazy*, *top-down* fashion, and prove its *soundness* and *completeness* for theories defined by (ground) *convergent* (terminating and ground confluent) rewrite systems.

As illustrated by the examples above, semantic unification (and equation solving) provides a method which cleanly integrates useful features of logic programming in a functional language. Furthermore, in theorem proving, it is often convenient (and necessary) to formulate deduction systems modulo specific equational theories, rather than treat these equations as primitive axioms (the theory consisting of the axioms for associativity and commutativity is a prime

example). In order to have a refutationally complete deduction system in such cases, complete sets of unifiers (in the theories under consideration) are required, as shown in [Plotkin, 1972].

The simpler variant of the problem, *semantic matching*, where one side of the goal is a ground (variable free) term, is also of interest. For example, if we could match with respect to the definitions of append and reverse, the function definition

$$\begin{aligned} palindrome(append(x, reverse(x))) &= true, \\ palindrome(append(x, a \cdot reverse(x))) &= true, \end{aligned}$$

could be applied to a term like $palindrome(1 \cdot 2 \cdot 1 \cdot nil)$ by finding that the pattern in the second definition matches the term when $x = 1 \cdot nil$ and $a = 2$. We describe syntactic restrictions on the equations under which simpler sets of transformation rules are sufficient for generating a complete set of semantic matchings.

Thereafter, we show that our first-order unification procedure, with slight modifications, can be used to solve the satisfiability problem in combinatory logic with a convergent set of algebraic axioms R, thus resulting in a complete higher-order unification procedure for R that retains the top-down and lazy features of the first-order procedure. Higher-order unification is of interest in its own right. Furthermore, it has applications in the areas of type-inferencing, higher-order reasoning, etc. Since the higher-order unification procedure outlined in this thesis is an extension of the one for the first-order case, including the top-down lazy feature, it should enjoy a reasonably fast implementation.

A large class of functions, including addition ($+$) and multiplication ($*$), satisfy the additional equations of associativity and commutativity (AC, for short), expressed by the following axioms (here f is the AC-function):

$$\begin{aligned} f(x, f(y, z)) &= f(f(x, y), z), \\ f(x, y) &= f(y, x). \end{aligned}$$

Therefore, it is important to have complete unification procedures in the presence of AC-functions. Notice that the second equation cannot be oriented as a rewrite rule without losing termination, and hence, we cannot directly apply the results for unification in convergent systems to this case. We deal with AC-functions in two stages. We show that a notion of *inductive*

simplification can be extended to handle *completely defined* AC-functions efficiently. However, the general case (that is, without the assumption of completely defined AC-functions) is more difficult, and a strict top-down approach does not work. We therefore provide a combined technique to solve the problem, wherein the AC-goals are delayed until some information is available about the corresponding subgoals.

Most of the completeness proofs described in this thesis require the equational theory to have a (ground) convergent presentation. Therefore, another relevant line of research is the formulation of orderings for termination proofs (since termination is essential in proving the existence and uniqueness of normal forms). Path orderings have been commonly used in theorem provers, even for AC-rewriting, despite the fact that they do not establish termination in the AC case. Furthermore, the associativity and commutativity axioms cannot be oriented without losing termination. Therefore, it is important to find orderings which can be used to prove termination of rewrite systems, modulo the combination of associativity and commutativity. We show that a simple (and easily implementable) restriction on the recursive path ordering [Dershowitz, 1982] is sufficient for establishing termination of extended rewriting, modulo associativity and commutativity.

It is well-known that any strategy for finding a complete set of unifiers (or matchings) for two terms with respect to a given theory may not terminate, even when the theory is presented as a finite and convergent set of rewrite rules [Heilbrunner and Hölldobler, 1987; Bockmayr, 1987; Dershowitz and Jouannaud, 1990; Jouannaud and Kirchner, 1991]. However, for some special classes of theories—associativity and commutativity, for instance—semantic unification is decidable. It is, therefore, of interest to find cases for which a particular complete procedure is provably terminating, thus implying that the semantic unification or matching problems in the corresponding theories are decidable. In particular, we study the restricted procedures for semantic matching and formulate different syntactic and semantic conditions (on the system of equations presenting the theory and the goals being solved) which result in decidability. Most of the decidable matching problems that we consider are finitary (that is, every matching problem has a finitely expressible set of solutions), and we show that a complete matching procedure is terminating in each case. We also consider decidable cases of semantic unification. We describe a notion of *flatness* on the right-hand sides of equations which results in decidability. However, unlike matching, the unification problem in this case is infinitary, and

therefore, the decision procedure illustrates that for flat systems, the generated (possibly infinite sets of) solutions form subsuming patterns. In most cases, we provide counterexamples to show that matching and unification become undecidable (as they usually are) when the conditions we propose are weakened.

The decidable matching problems considered in this thesis are useful in pattern-directed languages where a matching algorithm is required in order to mechanize pattern-directed invocations of functions. Furthermore, the decidability results formulated in this thesis would be useful in the areas of constraint solving and inductive theorem proving.

Previous approaches for semantic unification include narrowing and different variants thereof, such as "normalized" and "basic" narrowing; see, for example, [Fay, 1979; Hullot, 1980; Fribourg, 1985; Bosco *et al.*, 1987; Réty, 1987; Nutt *et al.*, 1989]. Martelli and Montanari [1982] used transformations on systems of equations to describe syntactic unification. The method was later adapted in [Martelli *et al.*, 1989] to provide a complete unification procedure for convergent rewrite systems. Furthermore, Gallier and Snyder have used transformations for describing equational and higher-order unification [Gallier and Snyder, 1989; Snyder and Gallier, 1989], while Kirchner [1984] uses the technique for unification in syntactic theories. Our method is a variant of narrowing, based on the top-down approach outlined in [Martelli *et al.*, 1989]. We achieve the effects of basic and normal narrowing in a lazy, top-down approach, and the introduction of directed goals (asymmetric goals, unlike the symmetric goals used, for example, by [Martelli *et al.*, 1989]) removes problems of generating extraneous reducible solutions.

A number of researchers have addressed the problem of unification in specific equational theories, for example, associativity [Plotkin, 1972; Makanin, 1977], associativity and commutativity [Stickel, 1981], and different variants of distributivity [Arnborg and Tidén, 1985; Contejean, 1992]; see [Siekmann, 1989; Jouannaud and Kirchner, 1991; Baader and Siekmann, 1993] for surveys on unification. Here, we study the unification problem with respect to convergent systems, where some of the function symbols in the system additionally satisfy the equivalences of associativity and commutativity.

We also study other specializations and extensions of the basic system of transformation rules, for example, for semantic matching and higher-order unification. Higher-order unification was initially studied by Huet [1975] and has been studied in the context of transformation rules

by Snyder [1990] and most recently by Dougherty and Johann [1992; 1993]. Our approach in this thesis is along the lines of the latter, using typed combinatory-logic as the formulation for higher-order systems.

Restricted versions of the problem of formulating decision procedures for special cases of matching and unification were looked at previously by Hullot [1980], Kapur and Narendran [1987] and Christian [1992].

Termination of a system of rewrite rules is important for using rewriting as a computational tool, and for simplification in theorem provers. Typically, termination proofs are done using *path orderings*, or by interpreting function symbols as multivariate polynomials; see [Dershowitz, 1987] for a survey of the area. In this thesis, we develop a precedence-based binary relation for proving termination of extended rewriting, modulo associativity and commutativity. Our ordering was inspired by the one in [Kapur *et al.*, 1990]. Similar research has been reported in [Bachmair and Plaisted, 1985; Bachmair, 1992], and recently in [Delor and Puel, 1993; Rubio and Nieuwenhuis, 1993].

1.2 Outline of the Thesis

In Chapter 2 we introduce most of the notations that we are going to use, and briefly provide some historical perspective on semantic unification. In Chapter 3 we introduce the systems of transformation rules for unification and matching, and prove their completeness. In Chapter 4 we provide a method for solving the unification problem for typed combinatory-logic, in the presence of a convergent set of algebraic axioms, while, in Chapter 5, we extend the method of Chapter 3 to theories containing associative and commutative functions. In Chapter 6 we discuss a precedence based relation for proving termination of AC systems. Chapter 7 provides results on different decision procedures for matching and unification. We conclude, in Chapter 8, with some open problems of interest.

2 BACKGROUND

In this chapter we introduce notations that we will use throughout this thesis. We also briefly recall important results about semantic unification.

2.1 Terminology

In this section we describe and review basic notation, and indicate some important results that are needed in the remainder of this thesis. For surveys of rewrite systems refer to [Huet and Oppen, 1980; Dershowitz and Jouannaud, 1990; Klop, 1992]. Most of the notations that we use in this thesis have been borrowed from [Dershowitz and Jouannaud, 1990].

Given a set $\mathcal{F}$ of function symbols and a (denumerable) set $\mathcal{X}$ of variables, the set of (first-order) terms $\mathcal{T}(\mathcal{F},\mathcal{X})$ is the smallest set containing $\mathcal{X}$ such that $f(t_1,\ldots,t_n)$ is in $\mathcal{T}(\mathcal{F},\mathcal{X})$ whenever $f \in \mathcal{F}$ and $t_i \in \mathcal{T}(\mathcal{F},\mathcal{X})$ for $i = 1,\ldots,n$. With every function symbol f we associate a unique natural number called the *arity*, which denotes the number of immediate subterms f can have in a well-formed term. Functions with arity 0 are called *constants*. Sometimes we use an infix notation for *binary* (2-ary) functions, that is, we write $x+y$ instead of $+(x,y)$. It is often convenient to consider $\mathcal{F}$ itself as being built of two sets, namely the set of *defined functions* (denoted $\mathcal{D}$) and the set of *constructors* (denoted $\mathcal{C}$), such that $\mathcal{C} \cup \mathcal{D} = \mathcal{F}$ and $\mathcal{C} \cap \mathcal{D} = \phi$. Variable free terms are called *ground*, for example $s(0)+0$. The set of variables in a term t is denoted as $\mathcal{V}(t)$. We say that a variable $x \in \mathcal{X}$ *occurs* in a term t, and write $occurs(x,t)$, if $x \in \mathcal{V}(t)$. A term t is said to be linear in a variable x if x occurs exactly once in t, while a term is *linear* if it is linear with respect to each of its variables, for example, $x+(s(y)*z)$. We will use $\equiv$ to denote syntactic identity of terms, to distinguish it from other forms of equality. Unless otherwise stated, we use the letters a through h for function symbols, x through z for variables and r through t for terms.

A term may be viewed as a finite ordered tree, the leaves of which are labeled with variables or constants, and the internal nodes of which are labeled with function symbols (of positive arity), with outdegree equal to the arity of the label. A *position* within a term may be represented—in Dewey decimal notation—as a sequence of positive integers, describing the

path from the outermost, "root" symbol to the head of the subterm at that position. Positions are also called *places* or *occurrences*. For example, in the term $f(g(h(x), y), h(b(a)))$ the subterm at position $2 \cdot 1$ is $b(a)$. By $t\mid_p$ we denote the *subterm* of t rooted at position p. A subterm of t is called *proper* if it is distinct from t.

Reasoning with equations requires replacement of subterms by other terms. A term t with its subterm $t\mid_p$ replaced by s is denoted by $t[s]_p$. We refer to any term u that is the same as t everywhere except below p (that is, $u[s]_p = t$, for some term s) as the context within which the replacement takes place.

A *substitution* is a special kind of replacement operation, uniquely defined by a mapping from variables to terms which is equal to identity almost everywhere, and written out as $\{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}$. Formally, a substitution σ is a function from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$, extended to a function from $\mathcal{T}$ to itself (also denoted as σ) in such a way that $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$, for each f and for all subterms $t_i, 1 \leq i \leq n$. We usually use lower-case Greek letters to denote substitutions. The *composition* of two substitutions σ and θ, denoted by $\sigma \circ \theta$ or simply $\sigma\theta$, is a composition of the two functions; thus, if $x\sigma = s$ for some variable x, then $x\sigma\theta = s\theta$. We say that substitution σ is *at least as general as* substitution θ (with respect to a set $\mathcal{X}' \subseteq \mathcal{X}$ of protected variables) if there exists a substitution τ such that $\sigma\tau = \theta$ (when σ and τ are restricted to $\mathcal{X}'$). We usually keep $\mathcal{X}'$ implicit, and write $\sigma \leq \theta$ to denote that σ is at least as general as θ.

A term t *matches* a term s if $s\sigma = t$ for some substitution σ; in this case we also say that t is an *instance* of s. For example, $0 + s(0)$ matches $x + y$ with the substitution $\{x \mapsto 0, y \mapsto s(0)\}$. A term s unifies with a term t if $s\sigma = t\sigma$, for some substitution σ. A substitution σ is called the *most general unifier* (mgu) of two terms s and t if, for any unifier θ of s and t, there exists a substitution τ such that $\theta = \tau\sigma$. The most general unifier of two terms is unique upto *variable renaming*. For example, the most general unifier of $x + y$ and $u + v$ (u and v are variables) is the substitution $\{x \mapsto u, y \mapsto v\}$. However, if the function symbol $+$ is also known to be associative and commutative then σ would be an *AC-unifier* of $x + y$ and $u + v$ if, after applying the unifying substitution, the two terms are equivalent (not identical) upto associativity and commutativity; for example, $\{x \mapsto v, y \mapsto u\}$ is also a solution in this case. AC-unification is a costly operation and, in general, it produces a finite complete basis set of unifiers [Stickel, 1981], which could potentially be very large.

An *equation* is an unordered pair of terms written in the form $s = t$, where either or both of s and t may contain variables, which are understood as being universally quantified. A set of equations E specifies an equational theory $=_E$ over the terms of $\mathcal{T}(\mathcal{F}, \mathcal{X})$, which can be defined using the notion of replacement ($\leftrightarrow$) based on the idea of "*replacement of equals for equals.*" Given a set of equations E, and two terms s and t, s is a replacement of t (denoted $s \leftrightarrow t$), if $s = u[l\sigma]_p$ and $t = u[r\sigma]_p$ for some context u, position p in u, equation $l = r$ or $r = l$ in E and a substitution σ. Intuitively, this stands for the replacement of an instance of one side of the equation by the corresponding instance of the other. For example, we have $0 + 0 \leftrightarrow 0$, with $\sigma = \{x \mapsto 0\}$.

The central idea of rewriting is to impose directionality on the use of equations in proofs. Unlike equations which are unordered, a *rule* over a set of terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is an ordered pair $\langle l, r \rangle$, usually written as $l \to r$. Rules differ from equations by their use. Like equations, rules are used to replace instances of l by corresponding instances of r; unlike equations, rules are not used in the reverse direction. A (finite) set of rules is called a *rewrite system* (or a *term-rewriting system*), and is usually denoted as R. A term s rewrites to another term t in one step, denoted $s \to t$, if for some rule $l \to r$ in R, position p in s and substitution σ it is the case that $l\sigma = s \mid_p$ and $t = s[r\sigma]_p$. In other words, the left-hand side (l) of the rule $l \to r$ matches a subterm of s, and t is the result of replacing this subterm by the corresponding instance of the right-hand side (r). We denote $s \to^* t$ when t is derivable from s, that is, when s rewrites to t in zero or more steps. In general, given any relation $\to$ we denote its inverse by $\leftarrow$, its symmetric closure ($\to \cup \leftarrow$) by $\leftrightarrow$, and its reflexive-transitive closure by $\to^*$. For the rewrite relation itself, we write $s \downarrow t$ if s and t *join*, that is, $s \to^* u$ and $t \to^* u$ for some term u. A term s is said to be *irreducible* or in *normal form* if there is no term t such that $s \to t$, and we use the notation $s_\downarrow$ to denote that s is in normal form. We write $s \to^! t$ if $s \to^* t$ and t is in normal form, and we say that t is the normal form of s. A rewrite rule $l \to r$ is *left-linear* if l is linear, and it is *right-linear* if r is linear. We say that a rewrite system is left- (right-) linear, if each of its rules is left- (right-) linear. We say that a rewrite rule $l \to r$ is *non-erasing* if every variable in l also occurs in r. A rewrite system is non-erasing if all its rules are non-erasing.

A rewrite relation ($\to$) is *terminating* if there exists no infinite chain of rewrites of the form $t_1 \to t_2 \to \ldots \to t_k \ldots$, that is, if its transitive closure ($\to^+$) is a *well-founded* ordering. A rewrite relation is (*ground*) *confluent* if, whenever two (ground) 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 must be a term v such that $s \rightarrow^* v$ and $t \rightarrow^* v$. A rewrite system which is both terminating and (ground) confluent is said to be (*ground*) *convergent*.

Given an equational theory E, we denote an *equational goal* as $s \stackrel{?}{=} t$, for terms s and t. We say that a goal $s \stackrel{?}{=} t$ has a *solution* σ if $s\sigma =_E t\sigma$. We usually deal with collections (multisets) of goals, and write them as $\{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$; a substitution is a solution to this collection, if it solves each of its subgoals. We are only interested in enumerating a *complete* set of solutions. Given a goal $s \stackrel{?}{=} t$, we say that a set $\mathcal{S}$ of solutions is complete if for every solution σ to $s \stackrel{?}{=} t$, there exists a θ in $\mathcal{S}$ such that θ is at least as general as σ (that is, $\exists\tau\forall x.x\theta\tau =_E x\sigma$).

Convergent rewrite systems are useful for equation solving. For a goal like $s(0)+x \stackrel{?}{=} s(s(0))$ the only solution of interest is $\{x \mapsto s(0)\}$. Solutions of the form $\{x \mapsto 0 + s(0)\}$, although valid when working with equational theories, are not in the simplest or irreducible form if we treat the program for addition as a convergent rewrite system. A solution is *irreducible* if it maps each of the variables in its domain to an irreducible term. For example, with the usual definition of $+$ over natural numbers, the substitution $\{x \mapsto y, z \mapsto s(0)\}$ is irreducible, while $\{x \mapsto 0 + 0\}$ is not. Thus, convergent rewrite systems allow a compact representation of a set of solutions to a goal. We will be interested only in irreducible solutions. Also, as we show later, rewrite systems allow us to work with *directed* goals, which provide more pruning capabilities.

Next, we consider some extensions of rewriting. A *conditional equation* is of the form

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

where $n \geq 0$. The first part consists of n equations of the form $s_i = t_i, 1 \leq i \leq n$, possibly containing variables. We can define the notion of "replacement" like in the case of unconditional equations. For example, if we consider the conditional equation given above and σ is a substitution such that $l\sigma = u\mid_p$ and $s_i\sigma \leftrightarrow^* t_i\sigma, 1 \leq i \leq n$, then $u[l\sigma]_p \leftrightarrow u[r\sigma]_p$. In other words, only those substitutions σ that are *feasible* (for which the condition can be proved recursively by a sequence of similar replacements) can be used to replace an instance of the left-hand side (l) by

the corresponding instance of the right-hand side (r). For example, considering the equations

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

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

Like in the case of unconditional equational theories we can talk of rewrite rules derived from conditional equations. Furthermore, similar concepts like ordering, termination and confluence also carry over to the case of conditional theories. For a detailed survey of conditional rewrite systems refer to [Sivakumar, 1989].

Conditional rules are also very important in equational programming. The rule

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

has only one premise in the condition. We can replace the left-hand side $\mathit{factorial}(x)$ by the right-hand side only for those substitutions for which the condition "holds." See [Dershowitz *et al.*, 1988] for different versions of the operational mechanism for checking if the condition holds. The simplest is to use rewriting itself to check that the terms in the condition have the same normal form. Without loss of generality, we assume, in the remainder of this thesis, that conditional rules are expressed as $c : l \rightarrow r$, where c is a equationally defined predicate.

In general, if we allow arbitrary equations as conditions, it is undecidable to even check if a conditional rule can be applied to a term. By restricting the terms in the condition to be "smaller" (in some well-founded ordering) we obtain a class of *decreasing* systems for which we have methods for checking important properties like confluence, etc., [Dershowitz *et al.*, 1987].

Definition 1 (Decreasing). A conditional rewrite rule is *decreasing*, if there is a well-founded extension $\succ$ of the proper subterm ordering that contains $\rightarrow$, such that for each rule $c : l \rightarrow r$ and any substitution σ, $l\sigma \succ c\sigma$.

Most useful functions (like *factorial*) can be defined using decreasing systems. Without this restriction of decreasingness, narrowing is not a complete strategy for solving equations in a conditional equational theory.

2.2 Previous Unification Methods

Semantic unification has been an active field of research over the last two decades. In this section, we give an overview of unification methods which relate to this thesis, namely:

- Semantic unification in convergent systems.
- Semantic unification in a general equational theory.

For completeness sake, we will also mention important results relating to unification in specific equational theories. We will express all the unification procedures in this thesis using transformation rules, as in [Martelli and Montanari, 1982; Kirchner, 1984].

2.2.1 Narrowing

Narrowing is complete for unification in convergent rewrite systems. The method uses (syntactic) unification (instead of matching) between the left-hand side of a rule and some subterm in one side of the goal. For the case of conditional narrowing, it must also be possible to extend this unifying substitution to be feasible for the equations in the condition.

Definition 2. A term s is said to *narrow* to another term t via a substitution σ, denoted as $s \leadsto^{\sigma} t$, if s has a non-variable subterm $s|_p$ which unifies via a most general unifier μ with the left-hand side l of a variant (after variable renaming) of a rule $c : l \to r$. Furthermore, τ is a substitution such that $c\mu\tau \downarrow \mathit{true}$, $\sigma = \mu \circ \tau$ and $t = s\sigma[r\sigma]_p$.

The transformation rules for conditional narrowing are shown in Table 2.1.

To apply this procedure we have a collection of equational goals G (initially consisting of the single equation $\{s \stackrel{?}{=} t\}$), which is transformed using the two transformation rules given in Table 2.1, until no further goals remain to be solved. We now provide some examples:

Example 1. Consider the convergent system of rewrite rules (R) for appending lists (lists are represented in the usual way, using the constant *nil* to denote the empty list, and the construct $x \cdot y$ to denote a list with *head* x and *tail* y):

$$app(nil, x) \quad \to \quad x \tag{2.1}$$

$$app(x \cdot y, z) \quad \to \quad x \cdot app(y, z) \tag{2.2}$$

Reflect	$\{s \stackrel{?}{=} t\} \cup G$
	$\leadsto$
	$G\sigma$
	where $\sigma = mgu(s,t)$
Narrow	$\{s \stackrel{?}{=} t\} \cup G$
	$\leadsto$
	$\{s[r]_p\mu \stackrel{?}{=} t\mu, c\mu \stackrel{?}{=} true\} \cup G\mu$
	where $c : l \rightarrow r$ is a renamed rule in R, and
	$\mu = mgu(l, s\mid_p)$

Table 2.1: Transformation rules for narrowing

We show a possible derivation sequence starting with the goal $app(app(x,y),z) \stackrel{?}{=} nil$:

$$\begin{array}{lll} \{app(\underline{app(x,y)},z) \stackrel{?}{=} nil\} & \leadsto_{\mathbf{Narrow(2.1)}} & \{app(y,z) \stackrel{?}{=} nil\}; \sigma = \{x \mapsto nil\} \\ & \leadsto_{\mathbf{Narrow(2.1)}} & \{z \stackrel{?}{=} nil\}; \sigma = \{x \mapsto nil, y \mapsto nil\} \\ & \leadsto_{\mathbf{Reflect}} & \phi; \sigma = \{x \mapsto nil, y \mapsto nil, z \mapsto nil\} \end{array}$$

where, $\mathbf{Narrow}(i)$ stands for narrowing using the i^{th} rule in R.

Here and elsewhere in this thesis, we use the format of the above example to show derivation sequences. Whenever there is a goal in the left-column, we would show the transformed set of goals (and substitutions, if need be) in the right-column, after the transformation rule mentioned in the middle column has been applied. If the first column for any line is empty, we apply the transformation rule to a subgoal on the right-hand column of the previous line. Finally, whenever multiple possible subgoals are available, we usually underline the one on which (or the position where) the named transformation rule was used.

Going back to Example 1, with the goal $app(app(x,y),z) \stackrel{?}{=} nil$, there exists infinite derivations using the simple narrowing strategy outlined in Table 2.1, for example:

$$\begin{array}{lll} app(x,y) \stackrel{?}{=} nil & \leadsto_{\mathbf{Narrow(2.2)}} & \{x_1 \cdot app(y_1,y) \stackrel{?}{=} nil\}; \{x \mapsto x_1 \cdot y_1\} \\ & \leadsto_{\mathbf{Narrow(2.2)}} & \{x_1 \cdot x_2 \cdot app(y_2,y) \stackrel{?}{=} nil\}; \{y_1 \mapsto x_2 \cdot y_2\} \end{array}$$

It is possible to formulate additional (pruning) rules, in this case, to eliminate this infinite branch. For example, we could use external information, such as a term of the form $x \cdot y$ can never be semantically unified with the constant nil, since there are no rules in R to transform either one of these symbols ($\cdot$ or nil) to the other. However, it is difficult to uniformly use pruning information with narrowing, since, in general, a narrowing based method has to explore all ways of applying rules to goals. The following example illustrates some of the problems:

Example 2. Let R be the convergent rewrite system:

$$f(x, a) \rightarrow 0 \tag{2.3}$$

$$g(b) \rightarrow 0 \tag{2.4}$$

Possible derivations for the goal $f(g(u), u) \stackrel{?}{=} 0$ are shown below:

$$\begin{array}{lll} \{\underline{f(g(u), u)} \stackrel{?}{=} 0\} & \leadsto_{\mathbf{Narrow(2.3)}} & \{0 \stackrel{?}{=} 0\}; \sigma = \{u \mapsto a, x \mapsto g(a)\} \\ & \leadsto_{\mathbf{Reflect}} & \phi; \sigma = \{u \mapsto a, x \mapsto g(a)\} \\ \{f(\underline{g(u)}, u) \stackrel{?}{=} 0\} & \leadsto_{\mathbf{Narrow(2.4)}} & \{f(b, b) \stackrel{?}{=} 0\}; \sigma' = \{u \mapsto b\} \\ & \leadsto & \mathbf{Fail} \end{array}$$

This example illustrates the fact that a simple restriction on narrowing, such as innermost narrowing, is incomplete. This is because if we would have narrowed only at the innermost position, then in the above diagram the first narrowing would be ignored resulting in declaring that the goal is unsatisfiable, which is incorrect. The problem occurs because the term $f(g(a), a)$ (that is, the term obtained from the left-hand side of the goal, with the solution applied to it) is irreducible at the $g(a)$ subterm. However, deterministically narrowing at the innermost position of the goal ($g(u)$) would only account for the situation in which this subterm would be reducible, and therefore would not generate the required solution. (It is easy to construct a similar example to show that outermost narrowing too, in general, is incomplete.) Notice that it is possible to use any particular position as a choice point in the narrowing tree, by considering the two cases, either the subterm at that position would eventually be reducible, or it would remain irreducible. Thus, one branch of the choice would explore different narrowings at any position, while the other branch would mark that position as irreducible. Completeness of this

kind of a narrowing strategy has been explored in [Bosco *et al.*, 1987], wherein an innermost order has been used for applying narrowings. In Chapter 3 we use a similar idea to provide a complete unification procedure for convergent systems; however, our solution looks at terms at the outermost position, and therefore allows for more pruning of unsatisfiable goals.

The narrowing strategy discussed above is complete for confluent rewrite systems (actually, narrowing is complete with respect to irreducible solutions for confluent and non-terminating systems). In fact, if the system is convergent it is possible to use simplification deterministically [Fay, 1979]. For such systems, another refinement of narrowing, which uses a notion of *basic* positions (a proper subset of all available positions) for applying rules from R, was proposed in [Hullot, 1980]. Basic narrowing is complete for convergent systems, but not for systems that are confluent and non-terminating; see [Middeldorp and Hamoen, 1992] for a counterexample.

Recently, narrowing based unification methods have been studied by way of their optimality. In general, unification in theories defined by convergent rewrite systems is an undecidable problem (see Chapter 7 for details), and therefore the usual notions of complexity cannot be used. However, completeness proofs for narrowing and other similar unification procedures (including the ones in this thesis) is based on the idea of lifting validity proofs to situations where terms may contain variables (with an appropriate substitution applied to them). Based on this observation, it is easy to see that whenever there are more than one possible validity proofs between any two given terms, all but one of these proofs may be ignored without sacrificing completeness of the resulting unification procedure. Bockmayr *et al.* [1992] have studied a restriction on narrowing which uses only the left-most innermost proofs, and is therefore optimal in the sense that two different narrowing derivations cannot generate the same narrowing substitution.

2.2.2 Top-Down Decomposition

A problem with narrowing is that it provides very little control on the positions where rules get applied; for a complete strategy, all possibilities have to be tried. Another approach for semantic unification, therefore, attempts to look at terms in a top-down (or almost top-down) fashion. In this section we briefly discuss the top-down strategy due to [Martelli *et al.*, 1989].

Martelli and others [1989] used an idea similar to syntactic unification to provide a system of transformation rules which is complete for unification with respect to convergent rewrite

systems. Their system of transformation rules appear in Table 2.2. We solve the goal from

Bind	$\{x \stackrel{?}{=} t\} \cup G$ $\rightsquigarrow$ $G\{x \mapsto t\}$ if x does not occur in t.
Decompose	$f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n)$ $\rightsquigarrow$ $s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n$
Mutate	$f(s_1, \ldots, s_n) \stackrel{?}{=} t$ $\rightsquigarrow$ $s_1 \stackrel{?}{=} l_1, \ldots, s_n \stackrel{?}{=} l_n, c \stackrel{?}{=} true, r \stackrel{?}{=} t$ where $c : l \rightarrow r$ is a renamed rule in R.
Expand	$x \stackrel{?}{=} t$ $\rightsquigarrow$ $x \stackrel{?}{=} t[r]_p, c\mu \stackrel{?}{=} true, l_1 \stackrel{?}{=} t_1, \ldots, l_n \stackrel{?}{=} t_n$ where $c : l \rightarrow r$ is a renamed rule in R, x occurs in t, and $t \mid_p \equiv f(t_1, \ldots, t_n)$.

Table 2.2: Transformation rules for top-down decomposition

Example 1 using this new strategy:

$$
\begin{array}{lll}
\{app(app(x,y),z) \stackrel{?}{=} nil\} & \rightsquigarrow_{\mathbf{Mutate(2.1)}} & \{app(x,y) \stackrel{?}{=} nil, z \stackrel{?}{=} x_1, x_1 \stackrel{?}{=} nil\} \\
& \rightsquigarrow^{*} & \{app(x,y) \stackrel{?}{=} nil\}; \sigma = \{z \mapsto nil, x_1 \mapsto nil\} \\
& \rightsquigarrow_{\mathbf{Mutate(2.1)}} & \{x \stackrel{?}{=} nil, y \stackrel{?}{=} y_1, y_1 \stackrel{?}{=} nil\}; \sigma \\
& \rightsquigarrow^{*} & \phi; \{x \mapsto nil, y \mapsto nil, z \mapsto nil, \ldots\}
\end{array}
$$

Using top-down decomposition, it is possible to have finite failure in this example, since a goal of the form $app(x,y) \stackrel{?}{=} nil$, when mutated using rule 2.2, would generate a new goal of the form $x_1 \cdot app(y_1, y) \stackrel{?}{=} nil$; at this point, the procedure would declare failure, since none of the transformation rules apply. There are, however, other problems with this method, since it uses goals symmetrically. First of all, whenever Expand is applicable, in effect, all possible narrowings of t have to be attempted, which could potentially be very expensive. Secondly, in Mutate, the left-hand sides from rules (the l_i subterms) are used symmetrically in the resulting subgoals. Therefore, subsequent mutation could apply rules into these subterms, which can

only generate reducible solutions, and worse still, may result in non-terminating sequences in situations which can be handled finitely by basic narrowing. These problems can be seen through the following example:

Example 3 ([Sivakumar, 1989]). Consider the convergent rewrite system given below:

$$f(a(x), b(x)) \rightarrow a(x) \tag{2.5}$$

$$a(s(x)) \rightarrow a(x) \tag{2.6}$$

$$b(s(x)) \rightarrow b(x) \tag{2.7}$$

Here, f, a and b are defined functions, while s is a constructor.

For this system, the goal $f(y, y) \stackrel{?}{=} y$ would stop with finite failure immediately, when using narrowing, since rule 2.5 is the only one which could be used, but is not applicable since $a(x)$ and $b(x)$ are not syntactically unifiable.

On the other hand, using the system of transformations described in Table 2.2, we have the following sequence:

$$\begin{array}{lll} f(y, y) \stackrel{?}{=} y & \leadsto_{\mathbf{Mutate(2.5)}} & \{y \stackrel{?}{=} a(x), y \stackrel{?}{=} b(x), a(x) \stackrel{?}{=} y\} \\ & \leadsto^{*} & a(x) \stackrel{?}{=} b(x) \end{array}$$

Thereafter, there will be an infinite failing sequence of derivation from this goal, using rules 2.6 and 2.7. The problem occurs because the subterms $a(x)$ and $b(x)$, introduced through the left-hand sides of a previously applied rule, are used for further mutations.

As mentioned in Section 2.2.1, for confluent and non-terminating theories, narrowing is complete with respect to irreducible solutions. Hölldobler [1987] developed a system, which is a variant of the one given in Table 2.2, for generating a complete set of solutions for this case (including the reducible solutions). This system consists of the transformations Bind, Decompose and Mutate from Table 2.2, together with the two new ones shown in Table 2.3.

Example 4. Consider the confluent system:

$$f \rightarrow c(f) \tag{2.8}$$

Imitate	$\{x \stackrel{?}{=} f(t_1, \ldots, t_n)\} \cup G$ $\leadsto$ $(\{x_1 \stackrel{?}{=} t_1, \ldots, x_n \stackrel{?}{=} t_n\} \cup G)\{x \mapsto f(x_1, \ldots, x_n)\}$ if x occurs in t, and, $x_1, \ldots, x_n$ are new variables.
Expand	$\{x \stackrel{?}{=} f(t_1, \ldots, t_n)\} \cup G$ $\leadsto$ $(\{r_1 \stackrel{?}{=} t_1, \ldots, r_n \stackrel{?}{=} t_n\} \cup G)\{x \mapsto l\}$ if x occurs in t, and $l \to f(r_1, \ldots, r_n)$ is a renamed rule in R.

Table 2.3: Transformation rules for confluent systems

We can now solve the goal $x \stackrel{?}{=} c(x)$, using *Expand*, to get the solution $\{x \mapsto f\}$.

Notice that narrowing would not generate the (reducible) solution $\{x \mapsto f\}$ in the example above.

2.2.3 General E-Unification

The notions of unification using transformations and top-down goal solving have been extended to handle the unification problem in a general equational theory, mainly through the work of Gallier and Snyder [1989; 1991]; we briefly review their system, which is shown in Table 2.4. Notice that goals are symmetric in this case, so, for example, elimination could be applied even when the right-hand side of a goal is a variable. For Lazy Paramodulation, whenever l is not a variable, we use the following variant:

$$\{s \stackrel{?}{=} t\} \cup G \rightsquigarrow \{l_1 \stackrel{?}{=} t_1, \ldots, l_n \stackrel{?}{=} t_n, s[r]_p \stackrel{?}{=} t\} \cup G,$$

where $s \mid_p \equiv f(t_1, \ldots, t_n)$ and $l \equiv f(l_1, \ldots, l_n)$.

Example 5 ([Snyder, 1991]). Let E be the equations:

$$x = f(g(x)) \tag{2.9}$$

$$g(h(y)) = g(k(y)) \tag{2.10}$$

$$g(k(f(z))) = z \tag{2.11}$$

Trivial	$\{s \stackrel{?}{=} s\} \cup G$
	$\rightsquigarrow$
	G
Eliminate	$\{x \stackrel{?}{=} t\} \cup G$
	$\rightsquigarrow$
	$G\{x \mapsto t\}$
	if x does not occur in t.
Decompose	$\{f(s_1, \ldots, s_n) \stackrel{?}{=} f(t_1, \ldots, t_n)\} \cup G$
	$\rightsquigarrow$
	$\{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\} \cup G$
Lazy Paramodulation	$\{s \stackrel{?}{=} t\} \cup G$
	$\rightsquigarrow$
	$\{s \mid_p \stackrel{?}{=} l, s[r]_p \stackrel{?}{=} t\} \cup G$
	where p is a non-variable position in s,
	$l \rightarrow r$ is a renamed rule in $E \cup E^{-1} \cup R \cup R^{-1}$.

Table 2.4: Transformation rules for general E-unification

We show some of the derivations starting with the goal $h(u) \stackrel{?}{=} u$, where u is a variable:

$$
\begin{array}{lll}
h(u) \stackrel{?}{=} u & \rightsquigarrow_{\mathbf{L.P.(2.9)}} & \{h(u) \stackrel{?}{=} x\,,\, f(\underline{g(x)}) \stackrel{?}{=} u\} \\
& \rightsquigarrow_{\mathbf{L.P.(2.11)}} & \{h(u) \stackrel{?}{=} x\,,\, g(x) \stackrel{?}{=} g(k(f(z))),\, f(z) \stackrel{?}{=} u\} \\
& \rightsquigarrow_{\mathbf{Eliminate}} & \{\underline{g(h(u))} \stackrel{?}{=} g(k(f(z))),\, f(z) \stackrel{?}{=} u,\, x \mapsto h(u)\} \\
& \rightsquigarrow_{\mathbf{L.P.(2.10)}} & \{g(h(u)) \stackrel{?}{=} g(h(y)),\, g(k(y)) \stackrel{?}{=} g(k(f(z))),\, f(z) \stackrel{?}{=} u,\, x \mapsto h(u)\} \\
& \rightsquigarrow^{*} & \{f(z) \stackrel{?}{=} f(z),\, x \mapsto h(f(z)),\, u \mapsto f(z),\, y \mapsto f(z)\} \\
& \rightsquigarrow_{\mathbf{Trivial}} & \{x \mapsto h(f(z)),\, u \mapsto f(z),\, y \mapsto f(z)\}
\end{array}
$$

In the above derivation we have used **L.P.** for lazy paramodulation. Each such step has been performed with a left-to-right orientation of an equation in E.

There have been different suggestions to improve the system of transformation rules provided in Table 2.4. For example, the fact that lazy paramodulation can be followed immediately by term decomposition can be iterated down, until at least one side of each of the equations is a variable. This action (called *top-unification*) reduces the positions where further paramodulation can be applied, and therefore results in a better system. See [Dougherty and Johann, 1990] for further details. Another suggestion, mentioned in [Hsiang and Jouannaud, 1988], is to consider more

cases for the terms on the two sides of the goals, and to perform variable bindings eagerly. Unfortunately, a proof of completeness of this system is still forthcoming (see [Jouannaud and Kirchner, 1991] for the actual system of transformation rules).

2.2.4 Semantic Unification in Specific Equational Theories

In this section we provide a brief summary of results for unification in specific equational theories. Almost all the material in this section has been adapted from [Jouannaud and Kirchner, 1991]. Other surveys of unification include [Siekmann, 1989; Baader and Siekmann, 1993].

$\mathrm{A}(f)$	Associativity	$f(f(x,y),z) = f(x,f(y,z))$
$\mathrm{C}(f)$	Commutativity	$f(x,y) = f(y,x)$
$\mathrm{Dr}(f,g)$	Right Distributivity	$f(g(x,y),z) = g(f(x,z),f(y,z))$
$\mathrm{Dl}(f,g)$	Left Distributivity	$f(z,g(x,y)) = g(f(z,x),f(z,y))$
$\mathrm{D}(f,g)$	Distributivity	$\mathrm{Dl}(f,g) \cup \mathrm{Dr}(f,g)$
$\mathrm{I}(f)$	Idempotence	$f(x,x) = x$

Table 2.5: Common equational axioms

Name	Decidable	Some References
Φ	yes	[Robinson, 1965; Paterson and Wegman, 1978; Martelli and Montanari, 1982]
$\mathrm{A}(f)$	yes	[Plotkin, 1972; Makanin, 1977]
$\mathrm{C}(f)$	yes	[Siekmann, 1979; Kirchner, 1986]
$\mathrm{I}(f)$	yes	[Hullot, 1980]
$\mathrm{A}(f)$,$\mathrm{C}(f)$	yes	[Stickel, 1981; Fages, 1984; Kirchner, 1989]
$\mathrm{A}(f)$,$\mathrm{I}(f)$	yes	[Siekmann and Szabó, 1984; Baader, 1986]
$\mathrm{C}(f)$,$\mathrm{I}(f)$	yes	[Jouannaud *et al.*, 1983]
$\mathrm{A}(f)$,$\mathrm{C}(f)$,$\mathrm{I}(f)$	yes	[Baader and Büttner, 1988]
$\mathrm{Dr}(f,g)$	yes	[Arnborg and Tidén, 1985]
$\mathrm{Dl}(f,g)$	yes	[Arnborg and Tidén, 1985]
$\mathrm{D}(f,g)$,$\mathrm{A}(f)$	no	Szabo
$\mathrm{D}(f,g)$,$\mathrm{A}(f)$,$\mathrm{C}(f)$	no	Szabo
$\mathrm{D}(f,g)$	unknown	Szabo

Table 2.6: Results on equational unification

In Table 2.5 we have listed the operators and equational theories that are of interested, while Table 2.6 contains the main results on unification using the equational axioms (and their

combinations) from Table 2.5. The main open problem in this area is determining whether $D(f,g)$ unification is decidable. A number of researchers have looked at different variants of the problem; see the surveys mentioned earlier for further references.

2.3 Discussion

In Section 2.2 we have presented some of the known results relating to semantic unification. We started with unification in confluent systems, in which case, narrowing is complete with respect to irreducible solutions, whereas the method of [Hölldobler, 1987] is complete even with respect to reducible solutions. For convergent systems, narrowing and variants of it, such as normalized and basic narrowing (and a combination thereof, due to [Nutt *et al.*, 1989]) are all complete strategies. However, we noted that none of these methods is completely satisfactory, since they have to apply rules non-deterministically at all possible positions, most of the time. The top-down solution proposed by [Martelli *et al.*, 1989] overcomes this difficulty, but has certain other problems of its own, since it uses goals in a symmetric manner. In this thesis we provide a complete unification method which combines the utilities of both narrowing and top-down decomposition while preserving the advantages of basic and normal narrowing.

We have briefly mentioned results on unification with specific equational theories for completeness. Although the approach developed in this thesis could be used to generate a complete set of unifiers for some of the equational theories mentioned in Section 2.2.4, our method would not necessarily result in a decision procedure (for example, consider associativity, unification for which has been proved to be decidable using techniques specific to the equational theory). Therefore, although our approach is somewhat more general from one perspective, unification in specific equational theories is of interest in its own right. We have also looked at previous results on general E-unification. Although we do not develop methods for general E-unification in this thesis, it is conceivable that the procedure outlined in this thesis can be extended for the general case; we outline a poossible approach in Chapter 8.

3 UNIFICATION WITH CONVERGENT SYSTEMS

In this chapter we describe a system of transformation rules for solving the satisfiability problem in an equational theory, and prove its completeness when the theory has a convergent presentation. We show that the transformation rules integrate the notions of "basic" and "normal" narrowing into a top-down, lazy approach. We also provide systems of transformation rules that are complete for matching problems in restricted equational theories.

3.1 Transformation Rules for Semantic Unification

From the definition of semantic unification, we see that a substitution σ is a unifier of two terms s and t in a theory E provided the terms $s\sigma$ and $t\sigma$ are equivalent in E (that is, $s\sigma =_E t\sigma$). Therefore, in order to generate a complete unification procedure, it is natural to start with a (decision) procedure for proving equality of terms (the *validity* problem) in the underlying theory. When the theory E enjoys a convergent presentation R, there exists a trivial decision procedure for the validity problem, namely, checking (syntactic) equality of normal forms, that is, $u =_R v$ if and only if $u \downarrow_R v$. Furthermore, in this case, any one complete strategy of reducing a term to its normal form is sufficient. In Table 3.1 we describe inference rules for proving validity using innermost reductions. In order to reduce a term to its normal form, we

Non-Top	$\dfrac{s_1 \rightarrow^! t_1, \ldots, s_n \rightarrow^! t_n, \forall l \rightarrow r \in R.\ \nexists \mu.\ f(t_1,\ldots,t_n) \equiv l\mu}{f(s_1,\ldots,s_n) \rightarrow^! f(t_1,\ldots,t_n)}$
Top	$\dfrac{s_1 \rightarrow^! t_1, \ldots, s_n \rightarrow^! t_n, l \rightarrow r \in R, f(t_1,\ldots,t_n) \equiv l\mu, r\mu \rightarrow^! t}{f(s_1,\ldots,s_n) \rightarrow^! t}$

Table 3.1: Validity using innermost reductions

first reduce all its arguments. Thereafter, there are two cases:

- A rule from R applies to the top-most position (that is, inference rule Top applies), in which case, we continue to reduce the right-hand side of the matching rule.

- If none of the rules from R are applicable at the top-most position, then the term is already in normal form.

The main idea behind using convergent rewrite systems for semantic unification is that if σ is a solution to $s \stackrel{?}{=} t$, then there must be a common normal form w such that $s\sigma \rightarrow^! w$ and $t\sigma \rightarrow^! w$. An equational goal $s \stackrel{?}{=} t$ can therefore be converted to two *directed* goals $s \rightarrow^? x$ and $t \rightarrow^? x$, where $x \in \mathcal{X}$ is a variable not in t or s. Another way to achieve the same effect is to introduce a new rule of the form $eq(x, x) \rightarrow true$ (where eq is a new function symbol and $true$ is a new constant), and consider matchings with respect to this augmented system. (Notice that whenever a rewrite system R is convergent, so is $R \cup eq(x, x) \rightarrow true$.) Therefore, solving $s \stackrel{?}{=} t$ in the original system is equivalent to solving the matching goal $eq(s, t) \rightarrow^? true$ in the new system. We will use these two conventions interchangeably in the remainder of this thesis.

Since we are interested in solutions with respect to a convergent rewrite system, we can further restrict the equation solving procedure to those solutions σ that correspond to innermost derivations $s\sigma \rightarrow_i^! w$ and $t\sigma \rightarrow_i^! w$ (notice that the particular choice of innermost derivations is for our convenience; it is sufficient to consider any one complete reduction strategy). We do not actually demand all solutions for completeness; rather, if $x\sigma$ and $x\tau$ are equal (in R) for all $x \in \mathcal{X}$ then (at least) one of σ or τ is deemed redundant.

We can use the transformation rules of Table 3.2 to solve the semantic unification problem with a convergent R. Some explanations are in order:

- Each transformation rule consists of an antecedent (the first line), a consequent (the third line) and, optionally, a condition (the fourth line, whenever present). Whenever a subgoal matches the pattern of the antecedent of a transformation rule, we can replace it with the corresponding consequent, provided the condition holds.

- The transformation rules given in Table 3.2 are non-deterministic, that is, for completeness all possibilities have to be tried. Thus, for any initial goal, we generate a tree (which we will call the *solution tree*) of such possibilities.

- We use expressions of the form $x \mapsto t$, where x is a variable, to keep track of partial solutions. An "unbound variable" is one that does not occur in the domain of the partial solution generated so far. Furthermore, in the transformation rule for mutation, we

Eliminate	$x \rightarrow^? t$ $\leadsto$ $x \mapsto t$ where x is an unbound variable that does not occur in t
Bind	$x \rightarrow^? s, x \mapsto t$ $\leadsto$ $x \mapsto s, mgu(s,t)$ if x does not occur in s
Mutate	$f(s_1, \ldots, s_n) \rightarrow^? t$ $\leadsto$ $s_1 \rightarrow^? l_1, \ldots, s_n \rightarrow^? l_n, r \rightarrow^? t, c \rightarrow^? true, l_1 \not\rightarrow, \ldots, l_n \not\rightarrow$ where $c : f(l_1, \ldots, l_n) \rightarrow r$ is a renamed rule in R
Decompose	$f(s_1, \ldots, s_n) \rightarrow^? f(t_1, \ldots, t_n)$ $\leadsto$ $s_1 \rightarrow^? t_1, \ldots, s_n \rightarrow^? t_n$
Imitate	$f(s_1, \ldots, s_n) \rightarrow^? x$ $\leadsto$ $s_1 \rightarrow^? x_1, \ldots, s_n \rightarrow^? x_n, x \mapsto f(x_1, \ldots, x_n)$ where x is an unbound variable, and $x_1, \ldots, x_n$ are new variables
Apply	$s \rightarrow^? t, \sigma$ $\leadsto$ $s \rightarrow^? t\sigma, \sigma$

Table 3.2: Transformation rules for semantic unification

have used irreducibility predicates: any collection of goals containing the irreducibility predicate $s \not\rightarrow$ has a solution σ provided $s\sigma$ is irreducible (more detailed definitions follow).

- We do not automatically apply resulting substitutions back into left-hand sides of goals. This gives basic-narrowing-like capabilities without having to keep explicit markers for basic positions [Hullot, 1980]. However, we now need an additional transformation rule (Imitate) to handle the situation in which the right-hand side of a goal is a variable.

- Although we show Apply as a transformation rule like the rest, we really require that the resulting substitutions be applied to the right-hand sides of goals eagerly, as mentioned in the proof of Theorem 3.

3.2 Correctness

We discuss properties of the system of transformations described in Section 3.1. In particular, we will be interested in showing that the procedure outlined in Table 3.2 indeed generates a *complete set* of semantic unifiers for convergent rewrite systems, and that mutation takes place only at *basic* positions in goals. We start with some definitions:

Definition 3 (Node). Let $G \equiv \{s_1 \rightarrow^? t_1, \ldots, s_n \rightarrow^? t_n\}$ be a set of goals, $\sigma \equiv \{x_1 \mapsto u_1, \ldots, x_m \mapsto u_m\}$ be a substitution and $\mathcal{P} \equiv \{u_1 \not\rightarrow, \ldots, u_k \not\rightarrow\}$ be a collection of irreducibility predicates. Then the collection

$$\{s_1 \stackrel{?}{\rightarrow} t_1, \ldots, s_n \stackrel{?}{\rightarrow} t_n, x_1 \mapsto u_1, \ldots, x_m \mapsto u_m, u_1 \not\rightarrow, \ldots, u_k \not\rightarrow\}$$

is called a *node*.

With G, σ and $\mathcal{P}$ as defined above, we write nodes as $\langle G; \sigma; \mathcal{P} \rangle$, or simply as $\langle G; \sigma \rangle$ when the irreducibility predicates are not important.

Definition 4 (Solution). A substitution θ is a *solution* to a node $\langle G; \sigma; \mathcal{P} \rangle$ if:

- θ extends σ, that is, $\theta = \sigma \circ \gamma$, for some substitution γ,
- $s_i\theta \rightarrow^! t_i\theta$, $1 \leq i \leq n$, and
- $u_j\theta$ is irreducible, for $1 \leq j \leq k$.

3.2.1 Soundness

In this section we provide a proof of soundness for the system of transformation rules described in Section 3.1.

Definition 5 (Soundness). A transformation rule is *sound* if its application does not introduce any new solutions, that is, if $N_i \rightsquigarrow N_{i+1}$ and σ is a solution to the node N_{i+1}, then σ must also be a solution to the node N_i.

Theorem 1 (Soundness). *All the transformation rules of Table 3.2 are sound.*

Proof. The proof is by inspection of all the transformation rules in question. In each case, we assume that σ is a solution to the consequent of the transformation rule and show that σ is also a solution to the antecedent, provided the conditions associated with the transformation rule hold. Furthermore, since R is convergent, we can assume σ to be irreducible without loss of generality.

Eliminate Since σ is known to be a solution to the consequent, we have $x\sigma \equiv t\sigma$. Furthermore, since σ is irreducible, $x\sigma$ is a normal form. Thus, $x\sigma \rightarrow^* t\sigma$ if and only if $x\sigma \equiv t\sigma$, which means that σ must also be a solution to the antecedent.

Bind The argument is similar to the case above. Notice that $x\sigma \equiv s\sigma \equiv t\sigma$ must hold, in order for σ to be a solution of the antecedent (that is, σ must be a unifier of s and t). In effect, this is what we have in the consequent, since we are introducing a most-general unifier.

Mutate Since σ is a solution to the consequent set of goals, we have $s_i\sigma \rightarrow^! l_i\sigma, 1 \leq i \leq n; c\sigma \rightarrow^! \mathit{true}$ and $r\sigma \rightarrow^! t\sigma$. From the following derivation we see that σ must also be a solution to the antecedent:

$$f(s_1, \ldots, s_n)\sigma \equiv f(s_1\sigma, \ldots, s_n\sigma) \rightarrow^* f(l_1\sigma, \ldots, l_n\sigma) \rightarrow r\sigma \rightarrow^! t\sigma.$$

Decompose Since σ is a solution to the consequent, we have $s_i\sigma \rightarrow^! t_i\sigma, 1 \leq i \leq n$. Furthermore, because of our formulation of the transformation rules and the initial matching goal (recall that we transform $s \stackrel{?}{=} t$ into $eq(s,t) \rightarrow^? \mathit{true}$, where eq is a new symbol) $f(t_1, \ldots, t_n)$ must be a subterm of some l_j introduced in a previous mutation step. Therefore, $f(t_1, \ldots, t_n)\sigma$ must be irreducible (because of irreducibility predicates introduced during mutation), and thus, $f(s_1, \ldots, s_n)\sigma \equiv f(s_1\sigma, \ldots, s_n\sigma) \rightarrow^! f(t_1\sigma, \ldots, t_n\sigma)$.

Imitate The proof is similar to the one for Decompose.

Apply In this case, we do not change any solutions, since we are applying the partial solution to a subgoal.

□

Notice that the irreducibility predicates are useful in proving soundness of decomposition and imitation, and not for mutation. The main purpose of these predicates is to allow only those solutions which correspond to an innermost proof; see Section 3.2.3 for further details.

3.2.2 Completeness

We now provide a completeness proof for the system of transformation rules under consideration. As in the proof of soundness of Section 3.2.1, we assume, without loss of generality, that all solutions are irreducible. We begin with the following definition of solution-preserving application of transformations, which is the inverse of a sound application:

Definition 6 (Solution Preserving). A transformation rule is *solution preserving* if its application conserves solutions, that is, if $N_i \equiv \langle G_i; \sigma_i \rangle \rightsquigarrow \langle G_{i+1}; \sigma_{i+1} \rangle \equiv N_{i+1}$ and σ is a solution to the node N_i, then σ must also be a solution to the node N_{i+1}.

Definition 7 (Completeness). Given an equational theory E and two terms s and t, whenever $s\theta =_E t\theta$, there is a derivation of the form

$$\langle s \stackrel{?}{=} t; \phi; \phi \rangle \rightsquigarrow^{!} \langle \phi; \mu; \mathcal{P} \rangle,$$

generating a solution μ, such that $\mu \leq_E \theta$ (that is, $\exists \tau \forall x. x\mu\tau =_E x\theta$).

Lemma 2. *The transformation rules* Bind *and* Eliminate *are solution preserving.*

Proof. Here we will assume that σ is a solution for the antecedent, and explain why σ must also be a solution for the consequent:

Eliminate By assumption, $x\sigma \rightarrow^{!} t\sigma$. However, since σ is a normalized substitution, it must be the case that $x\sigma \equiv t\sigma$, which is exactly what we have for the consequent.

Bind Since σ is a solution to the antecedent, we have $x\sigma \rightarrow^{!} s\sigma$. Furthermore, σ is a normalized substitution. Therefore, we have $x\sigma \equiv s\sigma$. Furthermore, $x\sigma \equiv t\sigma$ (since $\{x \mapsto t\}$ is a part of the partial answer for the antecedent). Thus, σ must be a unifier of s and t, and, in the consequent, we are introducing a most general unifier of the two terms into the substitution.

□

Theorem 3 (Completeness). *Let R be a decreasing conditional (ground) convergent rewrite system, and $G \equiv \{eq(u,v) \rightarrow^? true\}$ be a goal-set that admits a solution θ. Then, there exists a sequence of transformations, starting with the goal G (see Definition 7 above), which generates a solution μ which is at least as general as θ (that is, $\mu \leq_R \theta$).*

We will need the following ordering for the proof of completeness:

Definition 8 (Ordering $\succ_\theta$). Let $>$ be the smallest ordering on terms such that $s > t$ if and only if $s \rightarrow t$ or t is a proper subterm of s. We define the ordering $\succ_\theta$ on nodes, with respect to a solution θ: Let $\langle G_1; \tau_1 \rangle$ and $\langle G_2; \tau_2 \rangle$ be two nodes, each of which admit a solution θ; then, $\langle G_1; \tau_1 \rangle \succ_\theta \langle G_2; \tau_2 \rangle$ if and only if $\{s_1\theta, \ldots, s_n\theta\} \succ \{u_1\theta, \ldots, u_n\theta\}$, where $G_1 \equiv \{s_1 \rightarrow^? t_1, \ldots, s_n \rightarrow^? t_n\}$, $G_2 \equiv \{u_1 \rightarrow^? v_1, \ldots, u_m \rightarrow^? v_m\}$ and $\succ$ is the multiset extension of $>$.

Whenever $\rightarrow$ is well-founded, so are $>$ and $\succ$, and therefore $\succ_\theta$. Note that we have not used the irreducibility predicates in this definition. In fact, we do not need irreducibility predicates for the completeness proof.

Proof. For the proof, we assume that substitutions are applied eagerly to right-hand sides of goals, that is, we do not consider the transformation rule Apply in this proof.

Although we start with an empty substitution and the set of goals G as stated above, in general, we will have both a set of remaining goals and a partial computed solution for the current node under consideration. The proof proceeds by picking any subgoal $s \rightarrow^? t$ from the current collection, and showing that some solution preserving transformation rule applies to this goal. Furthermore, we argue that this application reduces the node in the ordering $\succ_\theta$. Reduction ensures that we eventually reach a node of the form $\langle \phi; \mu \rangle$. Finally, we show that the computed answer substitution μ is at least as general as the solution θ. We sketch the different aspects of the proof:

Reduction For this part, we consider different possibilities of the left-hand side subterm (s) of the selected subgoal ($s \rightarrow^? t$):

If s is a variable, then either Bind or Eliminate would apply. Each such application is solution preserving, by Lemma 2. There is also a reduction in the complexity with respect to $\succ_\theta$, since, in either case, the subgoal $s \rightarrow^? t$ is removed.

On the other hand, if s is not a variable, then we have a selected goal of the form $f(s_1, \ldots, s_n) \rightarrow^? t$, that is, $s \equiv f(s_1, \ldots, s_n)$. Since θ is a solution to this goal, we have $s\theta \rightarrow^! t\theta$. Furthermore, this is an innermost derivation sequence, that is,

$$s\theta \equiv f(s_1, \ldots, s_n)\theta \equiv f(s_1\theta, \ldots, s_n\theta) \rightarrow^* f(\hat{s_1}, \ldots, \hat{s_n}) \rightarrow^! t\theta,$$

where $s_i\theta \rightarrow^! \hat{s_i}, 1 \leq i \leq n$. We can distinguish two cases, depending on whether the last $\rightarrow^!$ (which we call the *choice* step below) consists of at least one rewrite step, namely:

- If the choice step is identity, then we know that $f(\hat{s_1}, \ldots, \hat{s_n}) \equiv t\theta$. We consider further cases, depending on the structure of t:

  Variable t In this case, Imitate would apply.

  Non-variable t The only possibility which can yield a solution is when $t \equiv f(t_1, \ldots, t_n)$, in which case, Decompose would apply. (If the leading function symbol of t is not f, the the goal $s \rightarrow^? t$ does not have a solution.)

  Each of these applications is solution preserving. For example, consider Decompose. Since θ is a solution to the antecedent, we have $f(s_1, \ldots, s_n)\theta \equiv f(s_1\theta, \ldots, s_n\theta) \rightarrow^! f(t_1\theta, \ldots, t_n\theta)$. Furthermore, this being an innermost derivation without a rule application at the top-most position ensures that $s_i\theta \rightarrow^! t_i\theta, 1 \leq i \leq n$, and thus this application of Decompose is solution preserving. Next, consider Imitate: If θ is a solution to the antecedent, we have $f(s_1, \ldots, s_n)\theta \equiv f(s_1\theta, \ldots, s_n\theta) \rightarrow^! x\theta$. Furthermore, since θ must be a normalized solution, $x\theta$ must be of the form $f(t_1, \ldots, t_m)$. Also, by assumption, no rule applies to the top-most position of $f(s_1\theta, \ldots, s_n\theta)$ in this derivation; therefore, we must have $s_i\theta \rightarrow^! t_i, 1 \leq i \leq n$. Finally, since $x_1, \ldots, x_n$ are new variables, we could set $x_i\theta \equiv t_i, 1 \leq i \leq n$, which shows that θ is a solution to the consequent.

Furthermore, in each case, there is a decrease in the complexity of the problem (with respect to $\succ_\theta$), because we replace $f(s_1, \ldots, s_n)$ by the multiset containing $s_i, 1 \leq i \leq n$.

- If the choice step consists of at least one rewrite, then some rule from R must be applicable at the top-most position of $f(\hat{s_1}, \ldots, \hat{s_n})$ (since each of its proper subterms is already in normal form). In this case, the transformation rule Mutate would apply to the goal. Suppose, $c : f(l_1, \ldots, l_n) \rightarrow r$ be the first applicable rule (from R) in the reduction $f(\hat{s_1}, \ldots, \hat{s_n}) \rightarrow^! t\theta$. Since this is a matching rule, and each $\hat{s_i}$ is in normal form, we must have $\hat{s_i} \equiv l_i\theta$. Furthermore, because the rule has a condition, which must hold for rule-application, we must have $c\theta \rightarrow^! \mathit{true}$. Finally, since the system is convergent, we must have $r\theta \rightarrow^! t\theta$. These arguments demonstrate that the application of Mutate under consideration is solution preserving.

  To establish reduction, we must compare $f(s_1, \ldots, s_n)\theta$ with the multiset consisting of $c\theta, r\theta$ and $s_i\theta, 1 \leq i \leq n$. We know that $f(s_1, \ldots, s_n)\theta \succ_\theta s_i\theta$ because $>$ has the subterm property, $f(s_1, \ldots, s_n)\theta \succ_\theta c\theta$ since $\rightarrow$ is decreasing conditional, and $f(s_1, \ldots, s_n)\theta \succ_\theta r\theta$ since $\rightarrow$ is well-founded and stable under substitution.

Generality As an outcome of the above discussion, we see that whenever θ is a solution to some initial equational goal, the transformation system eventually finds a node with a computed solution (in which there are no remaining subgoals to be solved).

That this computed solution is at least as general as the solution θ under consideration is a consequence of the fact that each application of a transformation rule is solution preserving.

□

A sketch of this completeness proof first appeared in [Dershowitz *et al.*, 1990]. However, in that paper, Definition 4.4 (Page 289) does not include the subterm property for the ordering $\succ_\theta$, which is incorrect (actually, the completeness proof in Section 5 of [Dershowitz *et al.*, 1990] uses the correct ordering). This was pointed out by Hanus [1993] and others.

3.2.3 Discussion

We have used the concept of irreducibility predicates in the transformation rules, which was used in the soundness proof, but not for completeness. The following example shows that irreducibility predicates are indeed necessary:

Example 6. Let R be the convergent rewrite system given below:

$$g(0,0) \rightarrow 0 \tag{3.1}$$

$$f(g(0,x)) \rightarrow g(x,0) \tag{3.2}$$

$$f(0) \rightarrow 0 \tag{3.3}$$

Here, f and g are defined functions, x is a variable and 0 is a constant. Consider the goal $f(g(y,y)) \stackrel{?}{=} 0$ (that is, $\{f(g(y,y)) \rightarrow^? x', 0 \rightarrow^? x'\}$, for some variable x'). One possible derivation sequence is shown below:

$$\begin{array}{lll} \{f(g(y,y)) \rightarrow^? x', 0 \rightarrow^? x'\} & \rightsquigarrow_{\textbf{Imitate}} & \{f(g(y,y)) \rightarrow^? x', x' \mapsto 0\} \\ & \rightsquigarrow_{\textbf{Mutate}} & \{g(y,y) \rightarrow^? g(0,x), g(x,0) \rightarrow^? x', x' \mapsto 0\} \\ & \rightsquigarrow_{\textbf{Decompose}} & \{y \rightarrow^? 0, y \rightarrow^? x, g(x,0) \rightarrow^? x', x' \mapsto 0\} \end{array}$$

Consider the last node, that is, $\{y \rightarrow^? 0, y \rightarrow^? x, g(x,0) \rightarrow^? x', x' \mapsto 0\}$, which has a solution $\sigma \equiv \{y \mapsto 0, x \mapsto 0\}$. However, this σ is not a solution for the antecedent (for the application of Decompose), since the term $g(0,0)$ (which appears on the right-hand side of a subgoal in this antecedent) is reducible, and thus, $g(y,y)\sigma \rightarrow^! g(0,x)\sigma$ does not hold. Notice that the irreducibility predicate for the application of Mutate using Rule 3.2 (that is, $g(0,x) \not\rightarrow$) would disallow this solution.

In Example 6, the problem occurred because there is a *critical overlap* (the left-hand side of Rule 3.1 unifies with a subterm of the left-hand side of Rule 3.2) in R; therefore, a potentially reducible term ($g(0,x)$) gets introduced into the right-hand side of a goal after mutation with the rule $f(g(0,x)) \rightarrow g(x,0)$. In fact, it is possible to incorporate special pruning rules to eliminate branches of the solution tree which contain goals with reducible right-hand sides. We will consider pruning rules in Section 3.3.3.

3.2.4 Basic Positions

The following definition of *basic positions* is a variation of the one used by Hullot:

Definition 9 (Basic Positions). Consider a term t and a set of non-variable positions U in t. Furthermore, if $t \rightsquigarrow_p t'$ using a rule $l \rightarrow r$, then the set of basis positions U' of t' is defined as:

$$U' = \{U - \{v \in U \mid p \preceq v\}\} \cup \{p \cdot v \mid v \in \overline{\mathcal{O}}(r)\}.$$

Here, p represents the occurrence in U where the rule $l \rightarrow r$ is applied, while $\overline{\mathcal{O}}(r)$ stands for the non-variable occurrences of r. In other words, whenever a step of $\rightsquigarrow$ is applied at any position p, we generate the new set of non-variable positions by replacing all positions below p with the non-variable occurrences from r.

Hullot showed that for a convergent rewrite system, narrowing applied only to basic positions is a complete strategy. A similar result holds for our system:

Lemma 4 (Basic Positions). *The unification procedure outlined in Table 3.2 applies rules only to basic positions in goals.*

Proof. Notice that we are dealing with unconditional rules in Definition 9 of basic positions. Thus, in Table 3.2 we do not consider the subgoal $c \rightarrow^? true$ generated by mutation.

The only place where we apply rules to goals is in the transformation for mutation. Potentially, any top-most position of a left-hand side of a goal is available for applying rules. However, since we have directed goals, the only terms that could appear on the left-hand sides of subgoals are either subterms of s or t (recall that we start with the single goal $eq(s,t) \rightarrow^? true$) or subterms of r, where r is a variant (after variable renaming) of a right-hand side of a rule (in R) previously used for mutation. (Furthermore, any time a variable occurs as the left-hand side of a goal, no further mutation would be possible for that goal.) Therefore, only basic positions could appear on the left-hand sides of goals. □

It is possible to extend Definition 9 to handle conditional rules also, by including non-variable positions from the condition. Lemma 4 would still be valid.

3.3 Refinements

Several refinements of the basic system of transformation rules developed in the previous sections are suggested here. Wherever necessary, a proof of completeness in presence of the additional transformation rule(s) is also outlined.

3.3.1 Normalized Goals

The concept of "normal narrowing" has been discussed in detail in the literature. Rewriting is a special case of narrowing, in which the corresponding narrowing substitution does not instantiate the goal variables. In normalized narrowing, whenever a term admits a rewrite step, such a step is taken deterministically, without considering any of the other possible narrowings of the term. Such a strategy is complete whenever the rewrite system is convergent [Fay, 1979]. In our system, we have to add the transformation rule described in Table 3.3 to have similar effects. Completeness, in presence of this additional transformation rule, can be proved using

Normalize	$s \rightarrow^{?} t, \sigma$
	$\rightsquigarrow$
	$\hat{s} \rightarrow^{?} t, \sigma$
	where $s\sigma \rightarrow^{+} \hat{s}$

Table 3.3: Transformation for normalizing

the following lemma:

Lemma 5 (Normalization Preserves Solutions). *Any application of Normalize is solution preserving. Furthermore, there is also a decrease in the complexity of the resulting goal with respect to the ordering* $\succ_\theta$.

Proof. Suppose θ is a solution to the antecedent. Thus, $s\theta \rightarrow^{!} t\theta$. Also, since θ is a solution, it must be an extension of σ. Thus, $s\sigma\theta \equiv s\theta$ and therefore one way to normalize $s\sigma\theta$ is to first rewrite $s\sigma$ to get $\hat{s}$, and then normalize $\hat{s}\theta$, and because R is convergent, the result should be the same, that is, $\hat{s}\theta \rightarrow^{!} t\theta$.

Decrease in $\succ_\theta$ can be shown by virtue of the fact that at least one rewrite step must take place from $s\sigma$ to $\hat{s}$. □

Since normalization causes a decrease in complexity for all solutions, and is solution preserving, it can be used in a deterministic fashion (unlike, say mutation, which has to be used in conjunction with decomposition or imitation) very much like normalized narrowing.

In the combined system, it is possible to apply rules to non-basic positions, since Normalize may result in putting some terms from σ into the left-hand side of the new subgoal (via $\hat{s}$). The following example will clarify the point:

Example 7. Consider the convergent rewrite system given below:

$$f(x,0) \rightarrow x \tag{3.4}$$

$$g(0) \rightarrow 0 \tag{3.5}$$

$$c(0,g(y),0) \rightarrow 0 \tag{3.6}$$

$$c(0,0,0) \rightarrow 0 \tag{3.7}$$

Here, f, g and c are defined functions, 0 is a constant, while x and y are variables. Considering the goal $c(f(x_1,x_2),x_1,x_2) \rightarrow^? 0$, we have derivations as shown below:

$$\begin{array}{lll}
\{c(f(x_1,x_2),x_1,x_2) \rightarrow^? 0\} & \leadsto_{\textbf{Mutate(3.6)}} & \{f(x_1,x_2) \rightarrow^? 0, x_1 \mapsto g(y), x_2 \mapsto 0\} \\
& \leadsto_{\textbf{Normalize}} & \{g(y) \rightarrow^? 0, x_1 \mapsto g(y), x_2 \mapsto 0\} \\
& \leadsto_{\textbf{Mutate(3.5)}} & \{y \rightarrow^? 0, 0 \rightarrow^? 0, x_1 \mapsto g(y), x_2 \mapsto 0\} \\
& \leadsto^* & \{x \mapsto g(0), y \mapsto 0\} \\
\\
\{c(f(x_1,x_2),x_1,x_2) \rightarrow^? 0\} & \leadsto_{\textbf{Mutate(3.6)}} & \{f(x_1,x_2) \rightarrow^? 0, x_1 \mapsto g(y), x_2 \mapsto 0\} \\
& \leadsto_{\textbf{Mutate(3.4)}} & \{x_1 \rightarrow^? x, x_2 \rightarrow^? 0, x \rightarrow^? 0, x_1 \mapsto g(y), x_2 \mapsto 0\} \\
& \leadsto & \textbf{Fail}
\end{array}$$

In the first part, we have used normalization, which we do not use it in the second. In the first part, we get a reducible (and hence redundant, since we have a convergent rewrite system) solution, while we obtain failure in the other case, since the two values of x_1 (0 and $g(y)$) are not (syntactically) unifiable, and Bind is the only possible transformation rule which could apply to this node. Note that there is a single normalized solution to the above problem ($\{x \mapsto 0, y \mapsto 0\}$) which is generated in either case using mutation of the initial goal with the rule $c(0,0,0) \rightarrow 0$. Furthermore, notice that in presence of the irreducibility predicates,

even the first branch will be eliminated, since the restriction associated with mutation using Rule 3.6 would dictate that $g(y)$, with the final solution applied to it, should be irreducible. Thus, although we may mutate in positions which are non-basic, we never, in fact, generate the corresponding reducible solutions, as shown by the following lemma:

Lemma 6 (Basic Solutions). *For the system of transformation rules under consideration, any branch which leads to a solution never uses mutation at a non-basic position.*

Proof. By Lemma 4 all transformation rules other than Normalize preserves basic positions on left-hand sides of goals. Furthermore, considering Normalize, because of our formulation of the transformation rules and the initial node, every variable in the domain of the partial solution σ must be bound to a term introduced through the left-hand side of a previous rule application using mutation, and must therefore have a corresponding irreducibility predicate associated with it. Therefore, any further mutation of such a term (or its subterms) would cause the irreducibility predicate to be violated and the solution to be discarded. □

Finally, when using Normalize, we cannot introduce irreducibility predicates, as shown by the following example:

Example 8 ([Réty, 1987]). Consider the convergent rewrite system:

$$g(a(x', y'), a(x', y')) \rightarrow f(h(x'), h(y')) \tag{3.8}$$

$$f(x'', x'') \rightarrow x'' \tag{3.9}$$

$$h(h(x)) \rightarrow x \tag{3.10}$$

We show the derivations corresponding to the goal $g(a(y, y), z) \rightarrow^? 0$ below:

$$\begin{array}{lll}
g(a(y,y),z) \rightarrow^? 0 & \rightsquigarrow_{\textbf{Mutate(3.8)}} & a(y,y) \rightarrow^? a(x',y'), z \rightarrow^? a(x',y'), f(h(x'),h(y')) \rightarrow^? 0 \\
& \rightsquigarrow^{*} & f(h(x'),h(y')) \rightarrow^? 0, y \mapsto x', x' \mapsto y', z \mapsto a(x',y') \\
& \rightsquigarrow_{\textbf{Normalize}} & h(y') \rightarrow^? 0, y \mapsto x', x' \mapsto y', z \mapsto a(x',y') \\
& \rightsquigarrow_{\textbf{Mutate(3.10)}} & y' \rightarrow^? h(x), x \rightarrow^? 0, y \mapsto x', x' \mapsto y', z \mapsto a(x',y') \\
& \rightsquigarrow^{*} & y' \mapsto h(0), y \mapsto h(0), x' \mapsto h(0), z \mapsto a(h(0),h(0))
\end{array}$$

Notice that if we were to impose an irreducibility requirement on the variable x'' (from Rule 3.9), then we would not get the solution, since the term matching x'' in this case is $h(h(0))$, which

is reducible. Réty also uses a similar solution when he extends the concept of basic positions to weakly-basic position; see [Réty, 1987].

3.3.2 Inductive Simplification

In Section 3.3.1 we have shown that it is possible to use normalization and have a complete unification strategy. The main advantage of using simplification (normalization) during unification is to keep goals reduced, and even eliminate infinite branches (see Example 11). In this section we deal with a special case of simplification, using *inductive consequences*, as defined below:

Definition 10 (Inductive Consequence). Given an equational theory E and two terms s and t, $s = t$ is an *inductive consequence* of E if and only if for any ground substitution σ there exists a proof of $s\sigma =_E t\sigma$.

A similar notion (using $\downarrow_R$ instead of $=_E$) can be defined in case E admits a convergent presentation R.

The main idea is the same as before. We want to replace a node of the form $\langle G; \sigma \rangle$ with a new node $\langle G'; \sigma \rangle$, such that each solution to the former is also a solution to the latter (using the concept of solution preserving transformation), and the complexity of the latter node (in the ordering $\succ_\theta$) is less than that of the former. Suppose $l = r$ is an inductive consequence of E. If we have a node $s \rightarrow^? t$ in G, such that a subterm of s at position p matches l with a substitution μ, then we replace the current goal with $s[r]\mu \rightarrow^? t$, provided there is a decrease in the ordering $\succ_\theta$. If θ is a ground solution of $s \rightarrow^? t$ (that is, $s\theta \rightarrow^! t\theta$), then $s[r]_p\theta =_E s[l]_p\theta \rightarrow^! t\theta$, which shows that the application of inductive simplification is solution preserving. The major advantage of inductive consequences is that they can be used deterministically for simplification, without having to use them for mutation, since the inductive consequences does not add any new ground proofs; see Example 11 below.

Inductive simplification is particularly useful for systems which obey the constructor discipline, that is, it is possible to partition the set of functions ($\mathcal{F}$) into two disjoint sets $\mathcal{C}$ and $\mathcal{D}$, such that the normal form of any ground term, with function symbols from $\mathcal{F}$, consists of symbols from $\mathcal{C}$ alone; the rules for addition and multiplication (see Example 11) defines such a system, in which $\mathcal{C} = \{0, s\}$ and $\mathcal{D} = \{+, *\}$.

3.3.3 Pruning Unsatisfiable Goals

Semantic unification is expensive to implement, because of the non-deterministic nature of the transformation rules given in Table 3.2. Furthermore, the unification problem in a convergent system is only semi-decidable: any complete strategy would enumerate a unifier, whenever one exists. However, on the negative side, if there is no unifier for a given set of equations, the complete strategy may not terminate—for instance, the unification problem in the theory of addition and multiplication (Rules 3.18 through 3.21 of Example 11) is known to be undecidable—see Chapter 7 for further details. Thus, one important criterion for judging the effectiveness of a unification procedure is to consider how often the procedure can successfully detect unsatisfiable goals.

We begin this section by showing that an extension to the top-down unification procedure outlined in Section 3.1 can eliminate unsatisfiable goals in situations in which narrowing would go into an infinite failing branch:

Example 9. Consider the convergent rewrite system

$$b(0,0) \rightarrow c \tag{3.11}$$

$$b(s(x),y) \rightarrow g(x,y) \tag{3.12}$$

$$g(s(x),y) \rightarrow g(x,y) \tag{3.13}$$

It is evident that the goal $b(y,1) \stackrel{?}{=} c$ has no solution (since if $y = 0$, the first rule does not apply, and for any other substitution for y, the normal form of $b(y,1)$ has either b or g as the root symbol). Applying narrowing to this goal would lead to an infinite failing branch (because of Rule 3.13). However, we could preprocess the rewrite system and make the observation that a term with g as the root symbol can never be unified with c. In our approach, starting with the goal $b(y,1) \rightarrow^? c$, after the initial mutation (decomposition would fail right away), we would get a subgoal like $g(x',y') \rightarrow^? c$. At this point, due to the above observation, we could fail the goal.

This example motivates the use of a special transformation rule for failing goals based on a *reachability graph* built from the rewrite system, which could eliminate a class of unsatisfiable goals. The reachability graph consists of nodes corresponding to the function symbols in the rewrite system. For every rule of the form $f(l_1,\ldots,l_n) \rightarrow x$ (where x is a variable), we draw a

directed arc from f to all the nodes in the graph, while for rules like $f(l_1, \ldots, l_n) \rightarrow g(r_1, \ldots, r_m)$, we draw an arc from f to g (see [Dershowitz and Sivakumar, 1988] for further details). Notice that the reachability test can be incorporated into the narrowing process too, with similar results. In fact, more recently, Chabin and Réty [1991] have developed an extension of the reachability test discussed here, for a narrowing based approach.

The unification procedure outlined in Section 3.1 also performs better than the one provided by Martelli and Montanari [1989] (one major difference between the two is that the latter uses goals symmetrically, and therefore generates reducible solutions), as can be seen from the following example:

Example 10. Let R be the convergent rewrite system given below:

$$f(g(x)) \rightarrow f(x) \tag{3.14}$$

$$g(f(x)) \rightarrow g(x) \tag{3.15}$$

$$f(f(x)) \rightarrow f(x) \tag{3.16}$$

$$g(g(x)) \rightarrow g(x) \tag{3.17}$$

Consider the goal $f(x) \stackrel{?}{=} g(x)$. Using normalized-narrowing (see [Nutt *et al.*, 1989] for details), we would get:

$$\begin{array}{lll} \underline{f(x)} \stackrel{?}{=} g(x) & \leadsto_{\mathbf{Narrow(3.14)}} & f(x') \stackrel{?}{=} g(g(x')), x \mapsto g(x') \\ & \leadsto_{\mathbf{Normalize}} & f(x') \stackrel{?}{=} g(x'), x \mapsto g(x') \end{array}$$

There are other ways to narrow the initial goal (we could either narrow $f(x)$ using Rule 3.16, or narrow $g(x)$ using Rules 3.15 or 3.17), however, each narrowing derivation leads to a similar subgoal; thus, there is an infinite failing branch.

Even when we use the approach outlined in [Martelli *et al.*, 1989], we have a similar problem:

$$\begin{array}{lll} f(x) \stackrel{?}{=} g(x) & \leadsto_{\mathbf{Mutate(3.14)}} & x \stackrel{?}{=} g(x'), f(x') \stackrel{?}{=} g(x) \\ & \leadsto & f(x') \stackrel{?}{=} g(g(x')), x \mapsto g(x') \\ & \leadsto & f(x') \stackrel{?}{=} g(x'), x \mapsto g(x') \end{array}$$

However, if we were to use the procedure outlined in this thesis, we get the following derivation (we would first transform the goal $f(x) \stackrel{?}{=} g(x)$ into two directed goals $f(x) \rightarrow^? y, g(x) \rightarrow^? y$, for a new variable y):

$$\begin{array}{lll} \underline{f(x) \rightarrow^? y}, g(x) \rightarrow^? y & \rightsquigarrow_{\textbf{Mutate(3.14)}} & \underline{x \rightarrow^? g(x')}, f(x') \rightarrow^? y, g(x) \rightarrow^? y, g(x') \not\rightarrow \\ & \rightsquigarrow_{\textbf{Eliminate}} & f(x') \rightarrow^? y, \underline{g(x) \rightarrow^? y}, g(x') \not\rightarrow, x \mapsto g(x') \\ & \rightsquigarrow_{\textbf{Normalize}} & f(x') \rightarrow^? y, g(x') \rightarrow^? y, g(x') \not\rightarrow, x \mapsto g(x') \end{array}$$

Since we have the irreducibility predicate $g(x') \not\rightarrow$, any further mutation of the subgoal $f(x') \rightarrow^? y$ (or $g(x') \rightarrow^? y$) is impossible, since such a mutation would cause $g(x')$ to become reducible. A similar situation would occur if we were to solve the initial goal in the other order, or if we were to mutate using a different rule from R, establishing that we can stop with finite failure. This example illustrates the importance of irreducibility predicates in pruning unsatisfiable goals (see Table 3.4 for the appropriate transformation rule).

We are now ready to formalize the failure cases as transformation rules, as shown in Table 3.4. Notice that the addition of failure rules do not cause any problem as far as soundness of the system is concerned (since failure terminates a branch without providing any solution). However, we would still require to show that completeness holds even when we have these additional transformation rules. Since the failure rules are deterministic, we can consider them one at a time, and argue that each one of them is solution preserving:

Theorem 7. *Each transformation rule in Table 3.4 is solution preserving.*

Proof.

OccurCheck If x occurs in t, then the goal $x \rightarrow^? t$ can have no finite solution.

Clash If σ is a solution to the antecedent, then, by definition, we should have $x\sigma \rightarrow^! s\sigma$ and $x\sigma \equiv t\sigma$. Furthermore, since we are only interested in irreducible solutions, we have $s\sigma \equiv t\sigma$, which is a contradiction to the fact that s and t are not syntactically unifiable. Thus, no such σ is possible.

Reachable Suppose θ is a solution to $f(s_1, \ldots, s_n) \rightarrow^? g(t_1, \ldots, t_m)$, that is, $f(s_1\theta, \ldots, s_n\theta) \rightarrow^! g(t_1\theta, \ldots, t_m\theta)$. Therefore, either some collapsing (rule which has a variable as the right-hand side) rule must have been used in this derivation, or a rule which has f and g as

OccurCheck	$x \rightarrow^? t$ $\leadsto$ **Fail** if x occurs in t
Clash	$x \rightarrow^? s, x \mapsto t$ $\leadsto$ **Fail** if s and t are not syntactically unifiable
Reachable	$f(s_1, \ldots, s_n) \rightarrow^? g(t_1, \ldots, t_m)$ $\leadsto$ **Fail** if there is no path from f to g in the reachability graph of R
Reducible	$t \not\rightarrow, \sigma$ $\leadsto$ **Fail** if $t\sigma$ is reducible

Table 3.4: Failure transformation rules

the root symbol of the left- and right-hand sides, respectively, must have been used. In either case, there should be a path from f to g in the reachability graph of R, which is a contradiction.

Reducible If θ is a solution to $\{t \not\rightarrow, \sigma\}$, then $t\theta$ must be irreducible. However, since $t\sigma$ is reducible, and θ must extend σ to be a solution, no such θ can exist.

□

3.4 Example

The following example illustrates all the notions developed so far:

Example 11. Consider the convergent rewrite system for addition ($+$) and multiplication ($*$) over natural numbers:

$$0 + x \rightarrow x \tag{3.18}$$

$$s(x) + y \rightarrow s(x + y) \tag{3.19}$$

$$0 * x \rightarrow 0 \tag{3.20}$$

$$s(x) * y \rightarrow y + (x * y) \tag{3.21}$$

We also use the following rules, which are provable inductive consequences of the above program, for simplification:

$$x + 0 \rightarrow x \tag{3.22}$$

$$x * 0 \rightarrow 0 \tag{3.23}$$

The four rule system (that is, Rules 3.18 through 3.21) given above is already complete and convergent (thus, the inductive consequences need not be used for mutation). Furthermore, the entire system, consisting of Rules 3.18 through 3.23, is terminating; this ensures that whenever we use an inductive consequence for simplification, we do indeed have a decrease in the complexity of the goals.

$\{\underline{x + y} \rightarrow^? 0\}$	$\rightsquigarrow_{\textbf{Mutate(3.18)}}$	$\{x \rightarrow^? 0, y \rightarrow^? x_1, x_1 \rightarrow^? 0\}$
	$\rightsquigarrow_{\textbf{Bind}^*}$	$\{x \mapsto 0, y \mapsto 0, x_1 \mapsto 0\}$
$\{\underline{x + y} \rightarrow^? 0\}$	$\rightsquigarrow_{\textbf{Mutate(3.19)}}$	$\{x \rightarrow^? s(x_2), y \rightarrow^? y_2, \underline{s(x_2 + y_2) \rightarrow^? 0}\}$
	$\rightsquigarrow$	**Fail**

Table 3.5: Solving goals of the form $x + y \rightarrow^? 0$

We want to solve the goal $x * y \rightarrow^? s(0)$ (this is, in effect, the same as solving $x * y \stackrel{?}{=} s(0)$). We start by showing, in Table 3.5, that a simpler goal, $x + y \rightarrow^? 0$ has a unique solution. As before, we underline the left-hand side of the subgoal whenever mutation is used, and mention the rule (in R) used in the process. We also make use of failure rules whenever a goal is known to be unsatisfiable, and mention the reason why such goals should be unsatisfiable. For example, failure, in the last branch, occurs because there are distinct constructors on the two sides of the goal $s(x_2 + y_2) \rightarrow^? 0$, which would mean that the goal is unsatisfiable.

Next, in Table 3.6, we show derivations corresponding to the initial goal $(x * y \rightarrow^? s(0))$. We get failure in the first branch since $s(0)$ cannot be the normal form of 0. Thereafter, in Table 3.7, we show the derivations corresponding to the only remaining subgoal, $\{y_2 + (x_2 * y_2) \rightarrow^? s(0)\} \cup$

$\{\underline{x * y} \rightarrow^? s(0)\}$	$\rightsquigarrow$**Mutate(3.20)**	$\{x \rightarrow^? 0, y \rightarrow^? x_1, \underline{0 \rightarrow^? s(0)}\}$
	$\rightsquigarrow$	**Fail**
$\{\underline{x * y} \rightarrow^? s(0)\}$	$\rightsquigarrow$**Mutate(3.21)**	$\{x \rightarrow^? s(x_2), y \rightarrow^? y_2, y_2 + (x_2 * y_2) \rightarrow^? s(0)\}$
	$\rightsquigarrow$**Bind***	$\{y_2 + (x_2 * y_2) \rightarrow^? s(0)\} \cup \sigma_1$

Table 3.6: Solving goals of the form $x * y \rightarrow^? 0$

σ_1. Finally, the derivation tree for the goal $\{x_2 * y_2 \rightarrow^? 0\} \cup \sigma_3$, is shown in Table 3.8. Thus,

$\{\underline{y_2 + (x_2 * y_2)} \rightarrow^? s(0)\} \cup \sigma_1$	$\rightsquigarrow$**Mutate(3.18)**	$\{y_2 \rightarrow^? 0, x_2 * y_2 \rightarrow^? x_3, x_3 \rightarrow^? s(0)\} \cup \sigma_1$
	$\rightsquigarrow$**Bind***	$\{x_2 * y_2 \rightarrow^? x_3\} \cup \sigma_1 \cup \{y_2 \mapsto 0, x_3 \mapsto s(0)\}$
	$\rightsquigarrow$**Normalize**	$\{\underline{0 \rightarrow^? s(0)}\} \cup \sigma_1 \cup \{y_2 \mapsto 0, x_3 \mapsto s(0)\}$
	$\rightsquigarrow$	**Fail**
$\{\underline{y_2 + (x_2 * y_2)} \rightarrow^? s(0)\} \cup \sigma_1$	$\rightsquigarrow$**Mutate(3.19)**	$\{y_2 \rightarrow^? s(x_4), x_2 * y_2 \rightarrow^? y_4,$ $s(x_4 + y_4) \rightarrow^? s(0)\} \cup \sigma_1$
	$\rightsquigarrow$**Bind**	$\{x_2 * y_2 \rightarrow^? y_4, s(x_4 + y_4) \rightarrow^? s(0)\} \cup \sigma_2$
	$\rightsquigarrow$**Decompose**	$\{x_2 * y_2 \rightarrow^? y_4, \underline{x_4 + y_4} \rightarrow^? 0\} \cup \sigma_2$
	$\rightsquigarrow$**Table 3.5**	$\{\underline{x_2 * y_2} \rightarrow^? 0\} \cup \sigma_2 \cup \{x_4 \mapsto 0, y_4 \mapsto 0\}$

Table 3.7: Solving goals of the form $y + (x * y) \rightarrow^? s(0)$

$\{\underline{x_2 * y_2} \rightarrow^? 0\} \cup \sigma_3$	$\rightsquigarrow$**Mutate(3.20)**	$\{x_2 \rightarrow^? 0, y_2 \rightarrow^? x_5, 0 \rightarrow^? y_4\} \cup \sigma_3$
	$\rightsquigarrow^*$	$\{x \mapsto s(0), y \mapsto s(0), \ldots\}$
$\{\underline{x_2 * y_2} \rightarrow^? 0\} \cup \{y_2 \mapsto s(0), \ldots\}$	$\rightsquigarrow$**Mutate(3.21)**	$\{x_2 \rightarrow^? x_6, y_2 \rightarrow^? y_6, y_6 + (x_6 * y_6) \rightarrow^? 0\}$ $\cup\{y_2 \mapsto s(0), \ldots\}$
	$\rightsquigarrow$**Bind***	$\{\underline{y_6 + (x_6 * y_6)} \rightarrow^? 0\} \cup \{y_6 \mapsto s(0) \ldots\}$
	$\rightsquigarrow$**Normalize**	$\{s(x_6 * s(0)) \rightarrow^? 0\} \cup \{y_6 \mapsto s(0) \ldots\}$
	$\rightsquigarrow$	**Fail**

Table 3.8: Solving goals of the form $x * y \rightarrow^? 0$

the only solution ($\{x \mapsto s(0), y \mapsto s(0), \ldots\}$) to the initial goal is obtained from the first branch of Table 3.8. At one point in Table 3.7 we assumed that the goal $x_4 + y_4 \rightarrow^? 0$ has a single solution, which was shown in the derivations of Table 3.5. The substitutions σ_1, σ_2 and σ_3 used

in the derivation sequence have the following bindings:

$$\begin{aligned}\sigma_1 &= \{x \mapsto s(x_2), y \mapsto y_2\}\\ \sigma_2 &= \{x \mapsto s(x_2), y \mapsto s(x_4), y_2 \mapsto s(x_4)\}\\ \sigma_3 &= \{x \mapsto s(x_2), y \mapsto s(0), y_2 \mapsto s(0), x_4 \mapsto 0, y_4 \mapsto 0, \ldots\}\end{aligned}$$

Thus, for the initial goal $x + y \rightarrow^? s(0)$, our system produces the single solution $\sigma \equiv \{x \mapsto s(0), y \mapsto s(0)\}$, and terminates in finite time (since all the branches are closed). Notice that without normalization, we would not get a finite solution tree, since a goal of the form $x * 0 \rightarrow^? s(0)$ (Table 3.7, first branch) would generate an infinite branch through successive mutations. Also, note that in order to use Normalize, an inductive consequence (Rule 3.23) is required.

3.5 Transformation Rules for Semantic Matching

Although in general semantic matching is as complex as semantic unification (for example, solving the goal $s \stackrel{?}{=} t$ in the convergent theory R is equivalent to solving the goal $eq(s,t) \rightarrow^? true$ in the theory $R \cup eq(x,x) \rightarrow true$, as discussed before), for a large class of systems, a simpler set of transformation rules provide a complete set of semantic matches. In this section we consider simplifications of the set of transformation rules provided in Section 3.1. First of all, we look at matching in theories defined by non-erasing convergent systems, using the transformation rules of Table 3.9. The main difference between this system and the one given in Table 3.2 is that Bind and Eliminate, in this case, only has to deal with ground terms; also, we do not need a transformation for imitation, and, as we show in the proof of Theorem 8, we can apply substitutions as required, and thus, do not require Apply.

The following theorem provides the completeness result:

Theorem 8 (Completeness). *Let $\mathcal{R}$ be a non-erasing (ground) convergent rewrite system. If the goal $s \rightarrow^? N$ has a solution θ (that is, $s\theta \rightarrow^! N$), then there is a derivation of the form*

$$\{s \stackrel{?}{\rightarrow} N\} \stackrel{!}{\rightsquigarrow} \mu,$$

such that μ is a substitution at least as general as θ.

Eliminate	$\{x \rightarrow^? N\}$
	$\rightsquigarrow$
	$\{x \mapsto N\}$
	where x is a free-variable, N is a ground normal-form
Bind	$\{x \rightarrow^? N, x \mapsto N\}$
	$\rightsquigarrow$
	$\{x \mapsto N\}$
	where N is a ground normal-form
Mutate	$\{f(s_1, \ldots, s_n) \rightarrow^? t\}$
	$\rightsquigarrow$
	$\{s_1 \rightarrow^? l_1, \ldots, s_n \rightarrow^? l_n, r \rightarrow^? t, c \rightarrow^? \mathit{true}\}$
	where $c : f(l_1, \ldots, l_n) \rightarrow r$ is a renamed rule in R
Decompose	$\{f(s_1, \ldots, s_n) \rightarrow^? f(t_1, \ldots, t_n)\}$
	$\rightsquigarrow$
	$\{s_1 \rightarrow^? t_1, \ldots, s_n \rightarrow^? t_n\}$

Table 3.9: Transformation rules for semantic matching with non-erasing systems

Proof. The proof is almost identical to the one for Theorem 3. The only difference is that we now have to use a selection strategy for picking subgoals to solve.

We show that we need only deal with subgoals which have ground normal forms on their right-hand sides. Consider a goal of the form $s \rightarrow^? N$, where N is a ground normal form. Decomposition would preserve the property (that is, decomposition would only generate subterms of N on the right-hand sides of subgoals). Consider mutation, which generates the subgoals $s_1 \rightarrow^? l_1, \ldots, s_n \rightarrow^? l_n, r \rightarrow^? N, c \rightarrow^? \mathit{true}$. We then solve the $r \rightarrow^? N$ subgoal first, to get a partial solution σ. Thereafter, we apply this σ to the right-hand sides of the subgoals $s_i \rightarrow^? l_i, 1 \leq i \leq n$. Since σ is a solution to $r \rightarrow^? N$, we have $r\sigma \rightarrow^! N$. Thus, σ must be a ground substitution, or else we would have a situation in which a non-ground term ($r\sigma$) would rewrite to a ground normal form (N) using only non-erasing rules, which is not possible. Finally, since the rule ($f(l_1, \ldots, l_n) \rightarrow r$) used for mutation is non-erasing (that is, r contains all variables that collectively appear in $l_1, \ldots, l_n$), each $l_i\sigma$ must be a ground term. The rest of the proof is quite similar to the one for Theorem 3. □

Next, we consider matching in theories defined by convergent, left-linear rewrite systems. A system very close to the one in Table 3.2 works for this case. The new system is given in Table 3.10. We now state the completeness theorem:

Eliminate	$\{x \to^? t\}$
	$\rightsquigarrow$
	$\{x \mapsto t\}$
	where x is a free-variable that does not occur in t
Bind	$\{x \to^? s, x \mapsto t\}$
	$\rightsquigarrow$
	$x \mapsto s, mgu(s,t)$
	if x does not occur in s
Mutate	$\{f(s_1,\ldots,s_n) \to^? t\}$
	$\rightsquigarrow$
	$\{s_1 \to^? l_1, \ldots, s_n \to^? l_n, r \to^? t, c \to^? true\}$
	where $c : f(l_1,\ldots,l_n) \to r$ is a renamed rule in R
Decompose	$\{f(s_1,\ldots,s_n) \to^? f(t_1,\ldots,t_n)\}$
	$\rightsquigarrow$
	$\{s_1 \to^? t_1, \ldots, s_n \to^? t_n\}$

Table 3.10: Transformation rules for semantic matching with left-linear systems

Theorem 9 (Completeness). *Let $\mathcal{R}$ be a left-linear (ground) convergent rewrite system. If the goal $s \to^? N$ has a solution θ (that is, $s\theta \to^! N$), then there is a derivation of the form*

$$\{s \stackrel{?}{\to} N\} \stackrel{!}{\rightsquigarrow} \mu,$$

such that μ is a substitution at least as general as θ.

We need the following lemmata for the proof.

Lemma 10. *Let $\mathcal{R}$ be a left-linear convergent rewrite system. Then, for the initial goal $\{s \to^? N\}$, where N is ground, if $G \cup \{t \to^? t'\}$ is the set of subgoals generated by the procedure at some point and x is a variable in t', then t' must be linear with respect to x; furthermore, x does not occur in any right-hand side of a subgoal in G.*

Proof. If $\tau \to^? t$ is a subgoal generated by the procedure for the initial goal $s \to^? N$, then the variables of t must come either from N or from the left-hand side of some rule in $\mathcal{R}$. For our case, N is ground, and $\mathcal{R}$ is left-linear, thus this variable cannot occur in any other right-hand side in the goal set, and again t itself must be linear in this variable. □

Lemma 11. *Let $\mathcal{R}$ be a left-linear convergent rewrite system. Then, in solving $\{s \to^? N\}$, a subgoal of the form $s \to^? x$ can be ignored without having to solve it any further.*

Proof. Let G denote the remaining set of subgoals when $s \rightarrow^? x$ is encountered. By Lemma 10, x cannot appear in any right-hand side in G. Furthermore, we always solve subgoals of mutation using the following strategy: solve the $r \rightarrow^? t$ subgoal first, to get a solution σ, and then solve the remaining subgoals $s_i \rightarrow^? l_i\sigma$ in any order. Therefore, if we ever encounter a goal of the form $s \rightarrow^? x$, then x must be a variable which does not require instantiation (that is, x does not appear anywhere in the remaining subgoals). Thus, any such goal is trivially solvable (that is, it has a solution for any substitution for the variables in s), and can be ignored. (As a result, all the variables in s would have indeterminate solutions, unless they appear on the left-hand side of some other goal which causes instantiation.) □

We now state the proof of the theorem:

Proof. Notice that the major difference between the transformation rules for semantic unification (Table 3.2) and those for matching in left-linear systems (Table 3.10) is that the latter does not have the rule for imitation. However, by Lemma 11, whenever a variable appears as the right-hand side of a goal, that goal is trivially solvable. Therefore, the remaining transformation rules are enough to enumerate a complete set of solutions of any matching goal in this case.

Note that, as in the case of non-erasing systems, we could apply partial solutions as required, and therefore do not have to explicitly consider Apply. □

3.6 Discussion

Most of the results given in Section 3.1 were first presented in [Dershowitz *et al.*, 1990], while the completeness of matching in the restricted case was proved in [Dershowitz *et al.*, 1992]. The main advantage of the method outlined in this thesis, over narrowing, is that our approach provides more control on positions where rules get applied to goals. Completeness of narrowing strategies (for example, outer narrowing [You, 1989] and innermost narrowing [Fribourg, 1985]) have been investigated. These strategies are complete only when the rewrite system has additional restrictions, over and above convergence. The method outlined in this thesis, on the other hand, is complete for any convergent presentation. Furthermore, we only use basic positions for rule application, and have deterministic simplification to reduce branching in the solution tree.

Methods based on top-down decomposition are known to be complete for convergent systems [Martelli *et al.*, 1989]. However, in this case, a different problem occurs because they use symmetric goals, thus allowing mutation of subterms introduced through the left-hand sides of previous rule applications (see Example 3). It is well-known that solutions generated thus are reducible, and therefore redundant, since we have a convergent system. We solve this problem by considering directed goals ($\rightarrow^?$) as opposed to symmetric goals ($\stackrel{?}{=}$), and using only the left-hand sides of goals for mutation. Furthermore, since we have directed goals, we have separated out more cases based on the structure of terms in goals, thus facilitating *eager variable binding*. For example, in our case, by virtue of Lemma 2, we know that Bind and Eliminate are solution preserving, and therefore we can deterministically bind variables whenever they appear as the left-hand side of a directed goal. (See Section 10 of [Gallier and Snyder, 1989] for a discussion of the problems of proving completeness in presence of eager binding if symmetric goals are used.) Therefore, in essence, our method combines the advantages of [Martelli *et al.*, 1989] and [Nutt *et al.*, 1989] in a single unified framework.

Gallier and Snyder [1989] have addressed a more general problem, that of unification in arbitrary equational theories, and, as a special case, they discuss the problem of unification in ground convergent rewrite systems. Their solution is based on the fact that any equational theory can be completed (using *unfailing completion* techniques developed by [Bachmair, 1987] and others) into a (possibly infinite) ground convergent rewrite system. So, one way to generate a complete set of unifiers is to work in this terminally ground convergent system, provided the rules of inference are powerful enough to generate all the *critical pairs* which are required in order to generate this convergent system. However, they have used symmetric goals, and therefore generate some redundant solutions. This was, in fact, pointed out in [Dougherty and Johann, 1990], wherein a more efficient system for general E-unification has been presented.

4 HIGHER-ORDER UNIFICATION

There have been different proposals to combine higher-order features with first order equational reasoning (including [Breazu-Tannen, 1988; Dougherty, 1991], and others). These proposals deal with the combination of lambda-calculus and a first-order equational theory. Recently, Dougherty and Johann ([Dougherty and Johann, 1992; Dougherty, 1993]) proposed a method for higher-order reasoning by transforming lambda-calculus terms to combinatory logic, that is, they use a combination of combinatory-logic with an equational theory as the formulation of higher-order reasoning. In [Dougherty and Johann, 1992], they also provide a complete set of transformations for solving the satisfiability problem in such a combined system. Some of the main advantages (pointed out in [Dougherty and Johann, 1992]) of using combinatory logic as the basis of higher-order unification (as opposed to the traditional lambda-calculus based methods such as [Snyder, 1990; Nipkow and Qian, 1991], etc.) are: it eliminates some technical problems associated with bound variables, allows easy incorporation of type-variables, and facilitates the use of substitution like in the first-order case. We extend the first-order unification procedure from Section 3.1 to develop a complete method for solving the satisfiability problem in (typed) combinatory logic, together with a set of algebraic axioms R, when R can be presented as a convergent rewrite system.

The main problem with the approach used of [Dougherty and Johann, 1992] is that narrowing as a method for solving equations provides very little control, since a complete method based on narrowing has to, in general, apply rules to all possible positions in terms. For the first order case, different refinements of narrowing have been suggested, which include Fay [1979] (normalized narrowing), Hullot [1980] (basic narrowing), Martelli et al. [1989] (top-down decomposition), Réty [1987] (basic-normal narrowing) and the one outlined in this thesis (which combines features of basic and normal narrowing into a top-down approach). In this chapter we combine the higher-order formulation of combinatory-logic with the first order top-down approach outlined in Chapter 3. Thus, the combination enjoys the following additional advantages (over [Dougherty and Johann, 1992]):

- It provides more control on positions where rules get applied. In general, we apply rules only to the top-most position in goals.
- It is possible to incorporate additional pruning rules (for example, reachability analysis of [Dershowitz and Sivakumar, 1987; Chabin and Réty, 1991]) directly.

4.1 Notations for Combinatory Logic

We begin with a brief discussion of notations pertaining to combinatory-logic. *Types* are formed by closing a set of *base types* (for example, integer and boolean) under the type forming operation $\alpha_1 \supset \alpha_2$ (for types α_1 and α_2).

We have a set of (typed) *variables* X, and a set of (typed) *constants*. An *atom* is either a variable or a constant. Terms are formed in the usual way: every constant and variable is a term, and, whenever $t_1, \ldots, t_n$ are terms and A an atom, $A\ t_1, \ldots, t_n$ is a term. We assume the constants $\mathcal{I}, \mathcal{K}$ and $\mathcal{S}$ (called *redex atoms*), given types as usual.

For $\mathcal{CL}$ (the simply typed combinatory-logic terms), the following convergent rewrite system (henceforth denoted C) defines *weak reduction*:

$$\begin{aligned} \mathcal{I}x &\rightarrow x \\ (\mathcal{K}x)y &\rightarrow x \\ ((\mathcal{S}x)y)z &\rightarrow (xz)(yz) \end{aligned}$$

Note that this rewrite system is terminating only for typed combinatory logic; in the untyped situation, we could have $\mathcal{SII}(\mathcal{SII}) \rightarrow \mathcal{I}(\mathcal{SII})(\mathcal{I}(\mathcal{SII})) \rightarrow^* \mathcal{SII}(\mathcal{SII}) \rightarrow \cdots$. We define *combinatory-R reduction* as $\rightarrow_C \cup \rightarrow_R$, which is convergent whenever R is a convergent rewrite system (using a similar result from [Breazu-Tannen, 1988]).

It is well-known that using combinatory reductions is not enough to capture equivalence of lambda terms. For example, $\mathcal{SK}$ and $\mathcal{KI}$ are distinct normal forms with respect to $\rightarrow_C$, though their translations to lambda-calculus are both equal to $\lambda y \lambda z.z$. However, it is possible to extend combinatory-R equality to capture equivalence of functional terms, by using the following rule of *extensionality*:

Infer $s = t$ if $sD = tD$, where D is a new constant.

A term is said to be *pure* if it does not contain any constant introduced by the extensionality axiom. We use the notation $s =_{RC} t$ (or say that s and t are RC-equal) to denote the equality of the lambda-calculus translations of s and t with respect to $\beta\eta R$ convertibility (which, by virtue of the above discussion, is identical to the equality induced by $C \cup R$ with extensionality).

In formulating rules for validity and higher-order unification, we deal with unordered-pairs of terms. A pair $s \stackrel{?}{=} t$ is *trivial* if $s \equiv t$, and is *RC-valid* if $s =_{RC} t$. A term-pair $s \stackrel{?}{=} t$ has a *solution* σ if $s\sigma =_{RC} t\sigma$. These notions can be extended to collections of pairs in the usual way; for example, we say that a collection is valid if and only if each of its pairs is valid.

4.2 Validity

In this section we describe an innermost reduction strategy which solves the validity problem with respect to combinatory logic terms. This reduction strategy serves as the basis for the higher-order unification procedure that we develop in a later section.

Given two terms s and t, we want to find if $s =_{RC} t$. The set of transformation rules (called IR, for innermost reduce) shown is Table 4.1 can be used to solve this problem. The transformation rules that we provide here are a refinement of the set RVT used by [Dougherty and Johann, 1992] (see Section 2.2). For simplicity of exposition we consider the rewrite system to be left-linear. However, techniques similar to those outlined in [Dougherty and Johann, 1992] could be used to incorporate non-left-linear rules.

In Table 4.1, we use C to denote the weak reduction rules of combinatory logic, while R stands for the convergent rewrite system. The only major difference between IR and RVT of [Dougherty and Johann, 1992] is in Reduce; herein we suggest that any rewrite step must be applied to some innermost position. Furthermore, our system allows additional arguments only when the corresponding terms are irreducible (with respect to $C \cup R$). The following lemma shows that IR captures the notion of RC-equivalence of combinatory logic terms:

Lemma 12. *Let R be a convergent left-linear rewrite system. If there exists a sequence of IR transformations starting with the pair $s \stackrel{?}{=} t$ which yields the collection of term-pairs G, that is,*

$$s \stackrel{?}{=} t \rightsquigarrow^{!}_{IR} G,$$

then $s =_{RC} t$ if and only if G is trivial.

Reduce	$s \stackrel{?}{=} t$
	$\leadsto$
	$s' \stackrel{?}{=} t$
	where $s \rightarrow_{C \cup R} s'$ is any *innermost* reduction
Decompose	$Fs_1 \ \ldots \ s_n \stackrel{?}{=} Ft_1 \ \ldots \ t_n$
	$\leadsto$
	$s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n$
	if $Fs_1 \ \ldots \ s_n$ and $Ft_1 \ \ldots \ t_n$ are in $C \cup R$ normal form, where F is a non-redex constant
Extend	$s \stackrel{?}{=} t$
	$\leadsto$
	$sD \stackrel{?}{=} tD$
	if s and t are in $C \cup R$ normal form, D is a new constant

Table 4.1: Transformation rules for innermost reduction (IR)

Proof. We indicate a straight forward proof of the lemma by constructing an equivalent RVT reduction sequence.

Note that IR is a restriction of RVT. Thus, from Theorem 2.5 of [Dougherty and Johann, 1992] it follows that whenever G is trivial, $s =_{RC} t$ must hold.

For the other part, suppose $s =_{RC} t$. Compare the IR transformation sequence under question with a possible RVT sequence. Furthermore, suppose that the first pair on which the two differ is $s' \stackrel{?}{=} t'$. For RVT, the applied transformation rule could have been either for adding argument or for reduction (C or R). In the latter case, at least one of s' or t' must still be reducible. Thus, in IR, the only applicable transformation rule must be Reduce, although at a different position as compared to RVT. However, since RVT is non-deterministic with respect to the position where reduction takes place, the same reduce could have been done first. It is then possible to construct a new RVT sequence which has one more step in common with the IR sequence under consideration. Similar arguments can also be provided if the transformation used by RVT was for adding an argument.

Thus, IR is sufficient to prove extensional (RC) equality between $\mathcal{CL}$ terms, with respect to a left-linear rewrite system R. □

Since IR is a restriction of RVT, we also have that every sequence of IR transformations is terminating (by virtue of a similar observations made about RVT in [Dougherty and Johann, 1992]). Typing plays an important role in this proof, since it disallows repeated applications of Extend (it is only sensible to add arguments to terms which have function types, and a finite number of these additional arguments would reduce such a term to one of a base type).

The following definition encapsulates a decision procedure for checking RC-validity:

Definition 11. To check validity of $s =_{RC} t$, where s and t are $\mathcal{CL}$ terms, we proceed as follows:

1. Use innermost-reduction to obtain normal forms u of s and v of t (that is, $s \rightarrow^! u$ and $t \rightarrow^! v$).
2. Use decomposition to reduce $u \stackrel{?}{=} v$ to non-trivial subgoals $\{l_i = r_i \mid i = 1, \ldots, m\}$.
3. Recursively check validity of $l_i D =_{RC} r_i D, i = 1, \ldots, m$.
4. Return "valid" if all resulting subgoals are trivial.

In the following section we provide a set of transformation rules for higher-order unification, that relies on Definition 11 for its proof of completeness.

4.3 Unification

In this section we formulate a RC-unification procedure based on the transformation rules given in Table 3.2. Given two combinatory-logic terms s and t, we want to find a complete set of their RC-unifiers; each RC-unifier being a solution σ as indicated before (that is, we want to enumerate all substitutions σ such that $s\sigma =_{RC} t\sigma$, with the understanding that whenever two substitutions are RC-equal, at least one of them is redundant). Since $=_{RC}$ has no known presentation as a convergent rewrite system (so, transformations from Section 3.1 cannot be used verbatim) we use the idea of combining the convergent relation $C \cup R$ with extensionality, as discussed in Section 4.1. Roughly, to decide if $s\sigma =_{RC} t\sigma$, we reduce $s\sigma$ and $t\sigma$ (using innermost reductions) with respect to $C \cup R$ as far as possible, and then invoke extensionality. We repeating this process until all term-pairs are trivial (see Definition 11).

Therefore, our top-down method for combining combinatory-logic with a first-order rewrite system proceeds as follows:

- Given a goal $s \stackrel{?}{=}_{RC} t$, break it into a set of two directed goals of the form: $\{s \rightarrow^? x, t \rightarrow^? y\}$, and solve for x and y, which are new variables. The interpretation of a goal of the form $u \rightarrow^? v$ is: Find a solution σ such that $u\sigma \rightarrow^!_{C \cup R} v\sigma$.

- If the terms bound to x and y are syntactically unifiable with most general unifier θ, return one solution as the composition of the partial substitution collected so far with θ (details given below).

- If unification fails, s and t may still be RC-unifiable, by virtue of extensionality. We apply the extensionality axiom as required, and continue goal-solving with respect to $C \cup R$.

We now provide details for the different aspects. For the first part, we use the system of transformations given in Table 3.2, with the understanding that the only non-zeroary function (for example, f in Decompose, Mutate, etc., of Table 3.2) is "function application." In the rest of our discussion on RC-unification, we will refer to this system of transformation rules as WR, for weak-R transformations. When using this system of transformations for higher-order unification, we do not deal explicitly with types. Nevertheless, every transformation rule must be type preserving. We also insist that only pure terms are bound to variables (in Eliminate and Bind). Furthermore, we would restrict our attention to left-linear rewrite systems alone. The more general case can be handled using ideas developed in [Dougherty and Johann, 1992].

The transformation rules in Table 3.2 form the crux of the higher-order semantic unification procedure, since they provide a strategy of unification with respect to weak-combinatory equality. However, weak combinatory equality is not identical to $=_{RC}$. We need to extend the rules for weak-equality to handle extensionality. In order to introduce extensionality, we provide a new set of transformation rules (called EXT) shown in Table 4.2. The rules are non-deterministic (for example, both Bind and Extend would apply to the goal $x \stackrel{?}{=} t$), and, as usual, all possibilities have to be tried for completeness.

We are now ready to specify the complete goal-solving procedure:

Definition 12. (RCU) To solve $s \stackrel{?}{=} t$ for $\mathcal{CL}$ terms s and t, we proceed as follows:

1. Use WR to obtain a partial solution $\{x \mapsto u, y \mapsto v\} \cup \sigma$ to the goal $\{s \rightarrow^? x, t \rightarrow^? y\}$.

2. Use EXT to reduce $u\sigma \stackrel{?}{=} v\sigma; \phi$ to subgoals $\theta; \{l_i \stackrel{?}{=} r_i \mid i = 1, \ldots, m\}$.

Bind	$\{x \stackrel{?}{=} t\} \cup E; G$ $\Rightarrow$ $\{x \mapsto t\} \cup E\{x \mapsto t\}; G\{x \mapsto t\}$ if x does not occur in t
Decompose	$\{Fs_1 \ \ldots \ s_n \stackrel{?}{=} Ft_1 \ \ldots \ t_n\} \cup E; G$ $\Rightarrow$ $\{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} s_n\} \cup E; G$ where F is any constant other than $\mathcal{I}, \mathcal{K}, \mathcal{S}$,
Extend	$\{s \stackrel{?}{=} t\} \cup E; G$ $\Rightarrow$ $E; G \cup \{sD \stackrel{?}{=} tD\}$ where D is a new constant, provided sD and tD are type-correct

Table 4.2: Transformation rules for extension (EXT)

3. Recursively, find solutions τ_i to $l_i \stackrel{?}{=} r_i, i = 1, \ldots, m$.

4. Return $\{x \mapsto u, y \mapsto v\} \cup \sigma \cup \theta \cup \tau_1 \cup \ldots \cup \tau_m$, provided it is a substitution (that is, it passes the "occur check").

Example 12. Let R be the rule $F0x \rightarrow x$, where 0 and x are, respectively, a constant and a variable of type integer.

We look for solutions to the goal $z \stackrel{?}{=} F0$. Since both z (a variable) and $F0$ are in $C \cup R$ normal form, we have $u\sigma \equiv z$ and $v\sigma \equiv F0$ in step 2 of RCU. Therefore, one solution to the goal is $\{z \mapsto F0\}$. However, there are other solutions, since we can add an argument to the two sides, and thereafter attempt to solve $zD \stackrel{?}{=} F0D$, that is, $zD \stackrel{?}{=} D$ (here D is a new constant of

type integer). Some possible derivations with this goal are:

$$
\begin{array}{rll}
zD \stackrel{?}{=} D & \rightsquigarrow_{\mathbf{Mutate}_{\mathcal{I}}} & z \mapsto \mathcal{I} \\
zD \stackrel{?}{=} D & \rightsquigarrow_{\mathbf{Mutate}_{\mathcal{K}}} & z_1 \stackrel{?}{=} D, z \mapsto \mathcal{K} z_1 \\
 & \rightsquigarrow & \mathbf{Fail} \\
zD \stackrel{?}{=} D & \rightsquigarrow_{\mathbf{Mutate}_{\mathcal{S}}} & z_1 D(z_2 D) \stackrel{?}{=} D, z \mapsto \mathcal{S} z_1 z_2 \\
z_1 D(z_2 D) \stackrel{?}{=} D & \rightsquigarrow_{\mathbf{Mutate}_{\mathcal{K}}} & z_1 \mapsto \mathcal{K} \\
z_1 D(z_2 D) \stackrel{?}{=} D & \rightsquigarrow_{\mathbf{Mutate}_{\mathcal{S}}} & z_3(z_2 D)(D(z_2 D)) \stackrel{?}{=} D, z_1 \mapsto \mathcal{S} z_3 \\
 & \rightsquigarrow & \mathbf{Fail}
\end{array}
$$

In the figure we have shown three different ways of mutating the goal $zD \stackrel{?}{=} D$, corresponding to the three rules in C. The first possibility yields a solution immediately ($\{z \mapsto \mathcal{I}\}$). In the second case, we get a subgoal of the form $z_1 \stackrel{?}{=} D$, which has no solution (we cannot use elimination, since it would not give a pure substitution. Also, we cannot add further arguments, since D is of type integer). In the last case we have further branches, only one of which yield a solutions, namely, $\{z \mapsto \mathcal{S}\mathcal{K}z_2\}$, which, in fact, is the same as the previous solution, since $\mathcal{S}\mathcal{K}x =_{RC} \mathcal{I}$, for any variable x. We get failure in the last branch, since we have a subterm of the form $D(z_2 D)$ which is not type correct, D being of integer type. Finally, we could mutate zD using the rule from R; this would repeat the solution $\{z \mapsto F0\}$. Therefore, we have exhausted all possibilities, and have a finite solution tree in this case.

4.4 Completeness

Our main result is completeness of the transformation rules for finding solutions to higher-order equations:

Theorem 13 (Completeness). *Let R be a left-linear convergent rewrite system. If θ is a RC-unifier for two terms s and t, then there exists a RCU-sequence that enumerates a solution σ that is at least as general as θ.*

Note that any RCU-sequence in general would consist of alternate applications of WR and EXT. Therefore, the completeness proof is a combination of the following lemmata:

Lemma 14. *Let R be a left-linear convergent rewrite system. Suppose θ is a solution to the goal $s \rightarrow^? t$ (i.e., $s\theta \rightarrow^!_{C \cup R} t\theta$), then there is a sequence of WR steps starting with the goal $s \rightarrow^? t$ that generates a substitution σ which is at least as general as θ.*

Proof. This lemma is an adaptation of the completeness proof given in Section 3.2.2, when the convergent rewrite system under consideration is $C \cup R$ (for a convergent R). Thus, this lemma follows as a corollary of Theorem 3. □

Lemma 15. *Let R be a convergent left-linear rewrite system. If s and t are terms in $C \cup R$ normal form, such that θ is a solution to $s \stackrel{?}{=} t$, then there exists an EXT derivation of the form*

$$s \stackrel{?}{=} t; \phi \Rightarrow^! \sigma; G,$$

such that $\sigma \cup \tau$ is at least as general as θ, where τ is a solution to the set of goals G.

Proof. We have to do a case analysis based on the structure of s and t, like in the proof of Theorem 3. Without loss of generality, we can assume that θ is irreducible (with respect to $C \cup R$). So, when considering a successful IR sequence for the validity-proof of $s\theta \stackrel{?}{=} t\theta$, the first step of the proof could either involve a decomposition or an extension. If we were to consider decomposition, there are further cases, depending on whether s or t is a variable (in which case, we would use Bind from EXT) or not (then we use Decompose in EXT also). On the other hand, if we were to Extend in IR, we could use Extend in EXT too. Therefore, for every possible step in IR, there is a corresponding step which could be performed in EXT. Finally, one has to show that for each non-deterministic step in EXT, at least for one of the choices, there is a decrease in the complexity measure for the proof (similar to the ordering $\succ_\theta$ used in the proof of Theorem 3). In this case, the appropriate ordering consists of an extension of the one used to show that all IR sequences are terminating (that is, the ordering used in [Dougherty and Johann, 1992] to show that all RVT sequences are terminating). □

Any innermost reduction steps can be simulated using WR (with respect to $C \cup R$). Furthermore, we explicitly add arguments (Extend) and use decomposition to expose inner positions where arguments need be added.

4.5 Discussion

The method for finding higher-order unifiers outlined in this chapter was presented in [Dershowitz and Mitra, 1993]. Higher-order semantic unification is of interest in automated theorem proving, type inferencing, higher-order extensions of logic-programming, etc. Our higher-order unification procedure performs better than one based on narrowing. In particular, we provide an example showing how infinitely many branches of the solution tree can be eliminated.

Example 13. Let R be the convergent rewrite system:

$$\begin{aligned} A(Sx) &\rightarrow AZ \\ B(Sx) &\rightarrow BZ \end{aligned}$$

Here A, B, Z and S are constants, while x is a variable. We also assume that the C rules are available as usual.

Note that a term with A at its head can never be RC-unified with a term with B as its head, since there are no rules to change one to the other. Now consider the goal: $A(xZ) \stackrel{?}{=} BZ$. Using the approach of [Dougherty and Johann, 1992], the subterm xZ could be narrowed indefinitely:

$$\begin{aligned} xZ &\rightsquigarrow x_1 Z(y_1 Z), x \mapsto (\mathcal{S}x_1)y_1 \\ &\rightsquigarrow x_2 Z(y_2 Z), x_1 \mapsto (\mathcal{S}x_2)y_2 \end{aligned}$$

However, using the approach outlined in this thesis, we first transform the goal to get $\{A(xZ) \rightarrow^? x', x'' \mapsto (BZ)\}$, since (BZ) is in ground normal form. We therefore have the following possible derivation sequences:

$$\begin{aligned} \{A(xZ) \rightarrow^? x'\} &\rightsquigarrow_{\textbf{Imitate}} \{xZ \rightarrow^? x_1, x' \mapsto Ax_1\} \\ &\rightsquigarrow \textbf{Fail} \end{aligned}$$

$$\begin{aligned} \{A(xZ) \rightarrow^? x'\} &\rightsquigarrow_{\textbf{Mutate}} \{xZ \rightarrow^? Sx_1, AZ \rightarrow^? x'\} \\ &\rightsquigarrow \{xZ \rightarrow^? Sx_1, x' \mapsto AZ\} \\ &\rightsquigarrow \textbf{Fail} \end{aligned}$$

Whenever we bind x' to any term which has A as the head, we can prune the corresponding branch (because of the observation about R made before); thus, the initial goal is unsatisfiable.

This example shows that for RC-unification, the top-down approach works better than one based on narrowing. Similar pruning capabilities of the top-down approach for the first-order case alone has been mentioned in [Dershowitz and Sivakumar, 1987; Dershowitz *et al.*, 1990], and they carry over to higher-order solving also. More elaborate pruning mechanisms have been studied for the first order case by Chabin and Réty [1991], where a graph of terms based on R and the goals under question has been used. Their approach is top-down, so we believe that it is possible to combine it with the higher-order capabilities developed here.

5 UNIFICATION IN ASSOCIATIVE-COMMUTATIVE THEORIES

In this chapter we show that the transformation system of Chapter 3 can be extended to handle theories with associative-commutative (AC) functions. We begin with a special class of AC-theories in which all the AC-functions are *completely defined* (that is, the normal form of any ground term containing AC-functions is a constructor only term), and show that for this restricted class, a complete set of unifiers can be generated without having to use the costly AC-unification operation. In fact, the resulting procedure works like the basic system of Chapter 3 on an extended rewrite system, which includes rules with commuted left-hand sides for AC-functions. However, for the general case, a strict top-down approach does not work. The problem occurs because a subgoal for an AC-goal may generate an AC-function (at the root position) after normalization, and therefore may require further rearrangement (using the AC-axioms) before a rule can be applied at the top-most position. Therefore, a priori distribution of subgoals for mutation or decomposition does not work. We show that it is possible to delay solving the AC-goals until information is available about the subgoals, and then apply rules at the top-most position.

5.1 Completely Defined AC Functions

We start by showing that the notion of inductive simplification can be used to provide a simple and efficient unification procedure if the AC-functions under consideration are completely defined (provided we are interested in completeness with respect to ground solutions only). These restrictions are reasonable in a programming language (as opposed to a theorem proving) environment. For example, consider functions like addition, multiplication and *greatest common divisor* (gcd) over integers.

For unification with AC-systems, where all the AC-functions are completely defined, we need only formulate an additional transformation rule to handle the case when an AC-function appears at the top-level on the left-hand sides of goals. This new transformation rule, called AC-Mutate, is shown in Table 5.1. For ease of expression, we have ignored the irreducibility

AC-Mutate	$f(s_1, s_2) \rightarrow^? t$ $\leadsto$ $s_1 \rightarrow^? l_1, s_2 \rightarrow^? l_2, r \rightarrow^? t, c \rightarrow^? \mathit{true}$ where $c : f(l_1, l_2) \rightarrow r$ or $c : f(l_2, l_1) \rightarrow r$ is a renamed rule in R, f is an AC-function.

Table 5.1: AC-Mutation for completely defined functions

predicates in the transformation rule. However, it is due to the irreducibility requirement that we do not have decomposition rules for AC-functions. Decompose or Imitate would result in a completely defined AC-function on the right-hand side of a goal. This would mean that for every ground substitution the right-hand side is reducible, and hence the corresponding solution is redundant. For similar reasons, we also do not have to consider mutation with rules (in R) which have AC-functions in a proper subterm of its left-hand side (since this eventually reducible subterm would appear on the right-hand side of the mutated sub-goal).

We assume that all AC-functions are kept in their usual binary form. Completeness can be proved by virtue of the following lemma:

Lemma 16. *Let R be a convergent rewrite system in which all AC-functions are completely defined, then the transformation rule of Table 5.1, together with those from Table 3.2 (used only for non-AC goals), is complete.*

Proof. Let $>$ be the smallest ordering on terms such that $u > v$ if and only if $u \rightarrow v$ (by AC-rewriting) or v is a proper subterm of u ($>$ is well-founded whenever $\rightarrow$ is). Let R^{comm} be $R \cup \{c : f(l_2, l_1) \rightarrow r \mid c : f(l_1, l_2) \rightarrow r \in R\}$, for every AC-function f. Notice that R^{comm} is terminating whenever R is (since if $u \rightarrow_{R^{comm}} v$, then $u \rightarrow_{AC/R} v$, and thus $u > v$). We show, by induction on the ordering $>$, that any ground term s can be reduced to its normal form by standard rewriting, without having to use the AC-axioms, by using the system R^{comm} to reduce s.

If s is a constant and $s \rightarrow s'$, then by inductive hypothesis, the proposition holds for s', and thus for s, since $s \rightarrow s'$ could not have used the AC-axioms. In the other case, suppose $s' \equiv f(s_1, \ldots, s_n)$ be the subterm to be rewritten first (if there is no such subterm, that is, if s is already in ground normal form, then by the assumption of the theorem, s cannot contain any

AC-function, which are assumed to be completely defined). If f is non-AC, then the proposition holds by using the inductive hypothesis for any rewrite sequence starting from s'. Finally, if f is AC (thus $n = 2$), we consider an innermost rewrite of s'. By the inductive hypothesis, the property holds for the normalization of s_1 and s_2. Now consider $f(s_1', s_2')$, where $s_1 \rightarrow^! s_1'$ and $s_2 \rightarrow^! s_2'$. Since f is completely defined, s_1' and s_2' must be constructor-only terms. Furthermore, $f(s_1', s_2')$ must be reducible in R. Considering the first step of this reduction, it could either have been a rule application from R, or may have been the invocation of the commutativity axiom for f. In either case, in R^{comm}, the term is directly reducible, and the lemma follows by applying the inductive hypothesis on the corresponding reduct.

The argument above shows that for any ground term s, it is possible to mimic the innermost AC-reduction sequence out of s by using only rules in R^{comm} (without having to use the axioms for associativity or commutativity). Furthermore, R and R^{comm} compute the same set of ground normal forms, and therefore, we could solve goals with respect to R^{comm} and still have a complete procedure. The rest of the proof is identical to that of Theorem 3, using R^{comm} as the convergent rewrite system, and noticing that we are interested in enumerating only ground solutions. □

As a consequence of the above lemma, it is sufficient to consider the extended rewrite system without the AC-axioms for the unification problem. Also, we do not require AC-unification in the process, since the only transformation rule which requires unification is Bind. However, as previously discussed, s and t in this case will not contain AC-functions (or else we would generate reducible solutions).

Note that, in Lemma 16, we only require that every term of the form $f(s_1', s_2')$, where s_1' and s_2' are ground normal forms and f is an AC-function, be directly reducible (in R^{comm}). Therefore, if any of the rules with commuted left-hand sides is an inductive consequence of R, then the rule is not required. For example, consider the rules $\{0 + x \rightarrow x, s(x) + y \rightarrow s(x + y)\}$. In this case, R^{comm} consists of the additional rules $\{x + 0 \rightarrow x, y + s(x) \rightarrow s(x + y)\}$, each of which is an inductive consequence of R.

5.2 General AC Theories

We now consider the situation in which certain AC-functions may not have the completely defined property. We show that this situation needs more than a trivial extension to the method developed so far, and present a complete unification procedure for this case. For the remainder of this section, we assume that terms are in flattened form, with respect to the AC-functions, that is, we convert terms like $f(f(a,b),c)$ into $f(a,b,c)$ for each AC-function f, by considering such functions to be variadic. We use capital letters to denote variables which range overs multisets of terms. Furthermore, we need extended rules (for example, the extended rule corresponding to $f(a,b) \to c$ is $f(a,b,X) \to t$, where t is the normal form of $f(c,X)$) for all the AC-rules of the system.

We first argue that a naive approach of trying all possible partitions for mutation (as shown in Table 5.2) does not work.

General AC-Mutate	$f(s_1,\ldots,s_n) \to^? t$ $\leadsto$ $f(S_1) \to^? l_1, \ldots, f(S_m) \to^? l_m, r \to^? t$ where $f(l_1,\ldots,l_m) \to r$ is a renamed rule in R, f is an AC-function, $\{S_1,\ldots,S_m\}$ is a partitioning of $\{s_1,\ldots,s_n\}$.

Table 5.2: AC-Mutation for completely defined functions

The counterexample below illustrates the difficulties involved:
Example 14.

$$\begin{aligned} a(c) &\to f(c,d) \\ b(c) &\to f(c,d) \\ f(c,c) &\to e \end{aligned}$$

Here a, b and f are defined functions (f is AC), while c, d and e are constructors. Consider the goal $\{f(a(x),b(y)) \to^? f(e,d,d)\}$. In order to get the solution $\sigma = \{x \mapsto c, y \mapsto c\}$, we have to first mutate the two immediate subterms ($a(x)$ and $b(y)$) of the left-hand side of the goal before we can apply the AC-rule ($f(c,c) \to e$). In fact, in order to apply the AC-rule, we have to pick parts out of the two subterms after mutation. Thus, there seems to be no easy way to

look at some AC-term and an AC-rule and decide whether the rule can be used to mutate the term.

In fact, a similar problem can occur with the transformation for Decompose, even when the AC-function is a constructor, as illustrated by the next example.

Example 15. Consider the one-rule system:

$$a(c) \quad \rightarrow \quad f(c,d)$$

Here a is the only defined function, f (AC) is a constructor, while c and d are constants. Considering the goal $f(a(z), g(f(c,c)), c) \rightarrow^? f(x, g(x), y)$, it is easy to see that the solution $\{x \mapsto f(c,c), y \mapsto d, z \mapsto c\}$ can be generated if we were to first mutate the $a(z)$ subgoal using the available rule; thereafter, we can AC-unify the two sides of the goal. Notice that, as in Example 14 above, we have to split the term generated from $a(z)$ in order to get this solution. Therefore, Decomposition by partitioning the subgoals of an AC-function would not work. Fortunately, as we prove later, this is the only potential problem with the method.

The following is an outline of a method for resolving the issue for Example 14, which we later generalize. There are essentially three steps as indicated below:

Abstract Subterms of the AC-term are abstracted out using new variables. For this example, we get the following two sets of subgoals (x_1 and x_2 are the new variables, A_0 is the initial set of AC-goals, while N_0 is the initial set of non-AC goals):

$$A_0 = \{f(x_1, x_2) \rightarrow^? f(e,d,d)\},$$
$$N_0 = \{a(x) \rightarrow^? x_1, b(y) \rightarrow^? x_2\}.$$

Solve Subgoals The only subgoals in this case are non-AC equations, therefore, we use the transformation rules as described in Section 3.1 to solve them. The relevant steps are shown below:

$$\begin{aligned} \{a(x) \rightarrow^? x_1, b(y) \rightarrow^? x_2\} \quad &\rightsquigarrow_{\mathbf{Mutate}} \quad \{b(y) \rightarrow^? x_2, x \mapsto c, x_1 \mapsto f(c,d)\} \\ &\rightsquigarrow_{\mathbf{Mutate}} \quad \{x \mapsto c, y \mapsto c, x_1 \mapsto f(c,d), x_2 \mapsto f(c,d)\} \end{aligned}$$

Solve AC At this stage all the arguments to the AC-term have been normalized; so, AC-rules may be applied. In the above example, we can instantiate the bindings for x_1 and x_2 in

the AC-term, and use the AC-rule to show that the left-hand side ($f(f(c,d),f(c,d)) \equiv f(c,c,d,d)$) rewrites to the right-hand side ($f(e,d,d)$). This generates the only solution $\{x \mapsto c, y \mapsto c\}$ to the initial goal.

Note that, in general, after the subterms have been solved, the left-hand side of the AC-goal may contain variables. In such cases we would have to use AC-unification with available rules, and continue goal solving using the corresponding right-hand sides (detailed descriptions follow).

Based on the example, we now outline the important features of our solution:

- Keep the AC and non-AC goals separate.
- The non-variable subterms of the left-hand sides of AC-goals are abstracted using new variables. In effect, the newly formed AC-goal is "frozen."
- Solve non-AC goals, corresponding to the frozen AC-goals. If no such goal exists, there must be some AC-goal for which the abstracted variables have been bound to fully normalized terms.
- Solve the AC-goals which have fully normalized subterms.

Next, we provide details for the different parts. In order to keep the AC and non-AC goals separate, we have to redefine a node:

Definition 13 (Node). A node is a collection $\langle A; G; \sigma \rangle$, where A is a set of AC-goals, G is a set of non-AC goals, and σ is a partial solution.

As before, given the goal $s \stackrel{?}{=} t$, we start with the initial node $\langle \phi; \{eq(s,t) \rightarrow^{?} true\}; \phi \rangle$. We do not deal with irreducibility predicates for simplicity. Furthermore, we only consider unconditional rules; the extension to decreasing conditional systems is easy, following the ideas developed for non-AC systems in Chapter 3.

We need several new transformation rules. First of all, we have to introduce the AC-goals in the transformation rules of Table 3.2. Furthermore, we would like to keep as much of the top-down feature as possible. Therefore, we abstract subterms of an AC-goal only when the AC-function appears at the top-level. In Table 5.3 we provide the transformation rule for abstracting subterms of an AC-goal. We use two parts of the node structure, one to keep track of the abstracted goals, and the other for goals which still have to be looked at. The following lemma provides some useful properties about AC-abstract:

AC-abstract	$\langle A; \{f(s_1, \ldots, s_n) \rightarrow^? t\} \cup G\rangle$
	$\rightsquigarrow$
	$\langle A \cup \{f(x_1, \ldots, x_n) \rightarrow^? t\}; \{s_1 \rightarrow^? x_1, \ldots, s_n \rightarrow^? x_n\} \cup G\rangle$
	where f is an AC-function,
	$x_1, \ldots, x_n$ are new variables.

Table 5.3: Abstracting AC-goals

Lemma 17. *Any sequence of AC-abstract steps terminates. Furthermore, if θ is a solution to a set of goals G (containing both AC and non-AC subgoals) then there is a sequence*

$$\langle \phi; G\rangle \rightsquigarrow^!_{AC-abstract} \langle A'; G'\rangle,$$

such that θ is also a solution to $\langle A'; G'\rangle$.

Proof. Termination is simple: every application of abstraction reduces the second part of the node in the multiset extension of the ordering that compares left-hand sides of goals using the subterm property.

Notice that the result of a sequence of AC-abstract steps starting from the node $\langle \phi; G\rangle$ as indicated above, would be to separate out the AC and non-AC subgoals in G, after suitably abstracting the AC-subgoals. Consider a single abstraction step. If θ is a solution to $f(s_1, \ldots, s_n) \rightarrow^? t$, then $f(s_1\theta, \ldots, s_n\theta) \rightarrow^! t\theta$. Since we have used new variables to abstract the subgoals of the AC-goal, we can extend θ (with slight abuse of notation, we will use θ for the extended substitution also), such that $s_i\theta \rightarrow^! x_i\theta, 1 \leq i \leq n$. Thus, $f(x_1\theta, \ldots, x_n\theta) \rightarrow^! t\theta$ follows, since the rewrite system is convergent. Therefore, each application of AC-abstract is solution preserving. □

Next, we provide transformation rules which abstracts subterms of an AC-goal as and when necessary (that is, whenever the corresponding AC-function appears at the root of the left-hand side of a subgoal), as shown in Table 5.4. Finally, we deal with AC-goals, using transformations from Table 5.5, wherein, we use the notion of a variable being *fully instantiated*, which we now define:

Decompose	$\langle A; \{f(s_1,\ldots,s_n) \rightarrow^? f(t_1,\ldots,t_n)\} \cup G; \sigma\rangle$ $\rightsquigarrow$ $\langle A \cup A'; G \cup G'; \sigma\rangle$ where $\{s_1 \rightarrow^? t_1, \ldots, s_n \rightarrow^? t_n\} \rightsquigarrow^!_{AC\text{-}abstract} \langle A'; G'\rangle$.
Imitate	$\langle A; \{f(s_1,\ldots,s_n) \rightarrow^? x\} \cup G; \sigma\rangle$ $\rightsquigarrow$ $\langle A \cup A'; G \cup G'; \sigma \cup \{x \mapsto f(x_1,\ldots,x_n)\}\rangle$ where $\{s_1 \rightarrow^? x_1, \ldots, s_n \rightarrow^? x_n\} \rightsquigarrow^!_{AC\text{-}abstract} \langle A'; G'\rangle$.
Mutate	$\langle A; \{f(s_1,\ldots,s_n) \rightarrow^? t\} \cup G; \sigma\rangle$ $\rightsquigarrow$ $\langle A \cup A'; G \cup G'; \sigma\rangle$ where $f(l_1,\ldots,l_n) \rightarrow r$ is a renamed rule in R, $\{s_1 \rightarrow^? l_1, \ldots, s_n \rightarrow^? l_n, r \rightarrow^? t\} \rightsquigarrow^!_{AC\text{-}abstract} \langle A'; G'\rangle$.

Table 5.4: Transformations for non-AC goals

Definition 14 (Fully Instantiated). A variable x is said to be *fully instantiated* with respect to a node $\langle A; G; \sigma\rangle$ if no variable in $x\sigma$ appears on the right-hand side of a subgoal in either A or G.

The following lemma states that a node always has a suitable goal to be solved next:

Lemma 18. *Let $\langle A; G; \sigma\rangle$ be a node with the AC-goal $f(x_1,\ldots,x_n) \rightarrow^? t$. If there exists an $x_i, 1 \leq i \leq n$, that is not fully instantiated with respect to this node, then either G is non-empty or A contains another goal which has fully instantiated subterms.*

Proof. Since x_i is not fully instantiated, $x_i\sigma$ must contain a variable (say x) which appears on the right-hand side of a subgoal. If this subgoal happens to be in G, we are done. Therefore, suppose the subgoal under question is of the form $g(y_1,\ldots,y_k) \rightarrow^? x$, where g is AC. In this case, we look at this new subgoal $g(y_1,\ldots,y_k) \rightarrow^? x$ and see if all its variables (that is, $y_1,\ldots,y_k$) are fully instantiated, and proceed again as above.

Notice that we cannot continue indefinitely, since there are only a finite number of subgoals in A, and subgoals cannot get repeated in the process, since we use new variables each time we abstract. Now consider the last such subgoal, say $g'(z'_1,\ldots,z'_l) \rightarrow^? z$. If any $z'_j, 1 \leq j \leq l$, is still not fully instantiated, then the variable must be on the right-hand side of a subgoal of G

Top-AC	$\langle A \cup \{f(x_1, \ldots, x_n) \rightarrow^? t\}; G; \sigma\rangle$ $\rightsquigarrow$ $\langle A \cup A'; G \cup G'; \sigma \cup \mu\rangle$ each of $x_1, \ldots, x_n$ is fully instantiated, $\mu \in mgu_{AC}(f(x_1, \ldots, x_n)\sigma, l)$ for a renamed rule $l \rightarrow r$ in R, where $\langle \phi; \{r \rightarrow^? t\}\rangle \rightsquigarrow^!_{AC\text{-}abstract} \langle A'; G'\rangle$.
AC-Imitate	$\langle A \cup \{f(x_1, \ldots, x_n) \rightarrow^? y\}; G; \sigma\rangle$ $\rightsquigarrow$ $\langle A; G; \sigma \cup \{y \mapsto f(x_1, \ldots, x_n)\}\rangle$ each of $x_1, \ldots, x_n$ is fully instantiated, y is a free variable.
AC-Bind	$\langle A \cup \{f(x_1, \ldots, x_n) \rightarrow^? t\}; G; \sigma\rangle$ $\rightsquigarrow$ $\langle A; G; \sigma \cup \mu\rangle$ each of $x_1, \ldots, x_n$ is fully instantiated, $f(x_1, \ldots, x_n)\sigma$ is in normal form, $\mu \in mgu_{AC}(f(x_1, \ldots, x_n)\sigma, t)$.

Table 5.5: Transformation rules for AC-goals

(if not, it has to be in A which would be a contradiction to the assumption that this is the last subgoal in any such chain), and therefore the result. □

We now indicate a proof of completeness for this new system of transformation rules. We extend the ordering $\succ_\theta$ to the new structure of nodes, by considering a substitution to be a solution only if it solves both the non-AC and the AC subgoals of a node. Similarly, we deal with a notion of solution preserving application of a transformation with regard to this new node structure.

Theorem 19 (Completeness-AC). *Let R be a convergent rewrite system. If $G \equiv \{eq(u, v) \rightarrow^? true\}$ be a goal that admits a solution θ, then, there exists a sequence of transformations, starting with the goal G, which generates a solution which is at least as general as θ.*

Proof. As before, in the general case, we pick a subgoal from the current node, and show that some solution preserving rule applies to it, and that the resulting node results in smaller

decrease with respect to the ordering $\succ_\theta$. Let $s \rightarrow^? t$ be the selected subgoal. There are several cases to consider:

Non-AC Goal If s has a root function which is non-AC, we apply one of the variants of the transformations from Table 3.2. Let $s \equiv g(s_1, \ldots, s_n)$. There are two further cases:

- Each $s_i, 1 \leq i \leq n$, is a non-AC term. In this case, the argument is identical to the completeness proof of Section 3.2.2, since there is no abstraction taking place.
- Suppose at least one immediate subterm of s (say s_i) has an AC-function at the root (and thus needs abstraction). Let $s_i \equiv g'(s'_1, \ldots, s'_k)$, g' is AC. Like in the proof of Theorem 3 we consider an innermost derivation of $t\theta$ from $s\theta$, and as before, we have to do a case analysis based on whether a rule applies at the top-most position of $s\theta$ in this reduction. For example, consider the case when no rule applies at this position. Furthermore, t could be either an unbound variable or a term of the form $g(t_1, \ldots, t_n)$ (these are the only choices, since we consider, like in the proof of Theorem 3, that substitutions are applied eagerly to the right-hand sides of goals). Consider the latter case, so that the Decompose rule from Table 5.4 applies. If θ is a solution to the antecedent, we have $f(s_1, \ldots, s_n)\theta \rightarrow^! f(t_1, \ldots, t_n)\theta$. On the other hand, for θ to be a solution to the consequent, θ must be a solution to A' and G', which follows from Lemma 17. Thus, this application of Decompose is solution preserving. The proofs of solution preservation of Imitate and Mutate are along similar lines.

  To show decrease with respect to the ordering $\succ_\theta$, one has to do a case analysis as in the previous part. We again consider Decompose as an example. Let the abstracted goals corresponding to s_i be $g'(x_1, \ldots, x_k) \rightarrow^? t_i, s'_1 \rightarrow^? x_1, \ldots, s'_k \rightarrow^? x_k$. Therefore, when comparing nodes for decomposition, we have to show that $s\theta \succ_\theta g'(x_1, \ldots, x_k)\theta$ (since $s\theta > s_i\theta \succ \{s'_1\theta, \ldots, s'_k\theta\}$, using the subterm property of $>$). Furthermore, we know that $s'_j\theta \rightarrow^! x_j\theta, 1 \leq j \leq k$. Therefore, $g'(s'_1, \ldots, s'_k)\theta \rightarrow^* g'(x_1, \ldots, x_k)\theta$, and thus, $g'(x_1, \ldots, x_k)\theta$ is no bigger than $g'(s'_1, \ldots, s'_k)\theta$ in $>$, and hence $s\theta > g'(x_1, \ldots, x_k)\theta$. Similarly, it is possible to show decrease for the other transformation rules.

AC Goal If s has an AC-function at the root, then by Lemma 18 either there is another non-AC goal, in which case we could solve that, or there is another AC-goal (say $s' \rightarrow^? t'$) where all the variables of s' are fully instantiated. In the latter case, we consider this new subgoal $s' \rightarrow^? t'$.

Therefore, it is sufficient to solve AC-goals that have fully instantiated variables. As before, we consider an innermost derivation of $s\theta \equiv f(s_1, \ldots, s_n)\theta$ to its normal form $t\theta$, and have two possibilities:

- At least one rewrite step takes place at the top-most position. The fact that subgoals from the AC-goals are solved before a rule is applied at the top-most position, each $s_i\theta, 1 \leq i \leq n$, is normalized. So, a rule from R must match $f(s_1, \ldots, s_n)$. In the transformations for semantic unification, this situation is handled by the rule for Top-AC, and the preceding argument shows that this application is solution preserving. Decrease with respect to $\succ_\theta$ can be shown by noting that the new abstracted goals cannot be any more complex than the measure for $r\theta$ in this ordering.
- If no rule is applied to the top-most position, $s\theta$ must be in normal form, and must be identical to $t\theta$ (modulo the AC axioms). This leads to two further cases, depending on the structure of t. If t is a variable, we could bind it to s to get the required θ, if not, we have to unify the two sides. In each case, there is a decrease since the subgoal $s \rightarrow^? t$ is removed.

□

5.3 Discussion

In Section 5.2 we used a combination of lazy top-down approach (for non-AC functions) and abstraction (for AC-functions) to solve the problem of semantic unification with respect to convergent theories. This method—of delaying AC-goals by abstracting and solving subgoals first—was initially presented in [Mitra and Sivakumar, 1991]. Notice that an extension of narrowing (using AC-unification and extended rules) would also provide a complete strategy for the semantic unification problem (see Section 3 of [Mitra and Sivakumar, 1991] for more details). In this section we provide an example to illustrate that the method outlined here performs better than the one based on narrowing.

In the method outlined in Section 5.2, we have simply "frozen" the AC-goals while we solve their corresponding subgoals. However, it is possible to use information about available AC-rules and the AC-goals when the non-AC goals are being solved, and in the process eliminate possibly infinite paths in the search space. The following example would illustrate the point:

Example 16. Let R be the convergent rewrite system defined below:

$$a(c) \rightarrow *(c,d) \tag{5.1}$$
$$b(h,h) \rightarrow c \tag{5.2}$$
$$b(x,s(y)) \rightarrow g(x,y) \tag{5.3}$$
$$g(x,s(y)) \rightarrow g(x,y) \tag{5.4}$$
$$*(x,x) \rightarrow e \tag{5.5}$$
$$*(e,d) \rightarrow h \tag{5.6}$$

Here $*$ is the only AC-function; a, b and g are defined functions; c, d, e and h are constructors.

Consider the goal $*(a(b(x,y)),c) \rightarrow^? h$. Notice that a method based on narrowing would not terminate in finite time for this goal, because of the $b(x,y)$ subterm. Using abstraction, we would first transform this goal into AC and non-AC subgoals:

$$A_0 = \{*(x',c) \rightarrow^? h\},$$
$$N_0 = \{a(b(x,y)) \rightarrow^? x'\}.$$

We do not abstract the second subterm of the AC-goal, since it is already in normal form. Next, we attempt to solve the non-AC goal. The relevant derivations are shown in Table 5.6. Notice that we get failure in the first branch since x' gets bound to $a(x_1)$, and thereafter the AC abstracted subgoal $*(x',c) \rightarrow^? h$ is not satisfiable any more, because neither Rules 5.5 nor 5.6 would be applicable thereafter. In this sense, we can use the AC-goals as constraints while solving the non-AC goals. The only solution is generated by applying σ from the second branch to the left-hand side of the AC-goal, and solving it. Finally, the last branch does not generate a solution since it is evident from the rewrite system that a term with g at its root can never generate a c. This notion of *reachability* analysis was first introduced in [Dershowitz and Sivakumar, 1987; Dershowitz and Sivakumar, 1988], and was extended in [Chabin and

$a(b(x,y)) \to^? x'$	$\rightsquigarrow$**Imitate**	$b(x,y) \to^? x_1, x' \mapsto a(x_1)$
	$\rightsquigarrow$**Constraint**	**Fail**
$a(b(x,y)) \to^? x'$	$\rightsquigarrow$**Mutate(5.1)**	$b(x,y) \to^? c, *(c,d) \to^? x'$
	$\rightsquigarrow^*$	$b(x,y) \to^? c, x' \mapsto *(c,d)$
$b(x,y) \to^? c, x' \mapsto *(c,d)$	$\rightsquigarrow$**Mutate(5.2)**	$\{x \mapsto h, y \mapsto h, x' \mapsto *(c,d)\} = \sigma$
$b(x,y) \to^? c, x' \mapsto *(c,d)$	$\rightsquigarrow$**Mutate(5.3)**	$x \to^? x', y \to^? s(y'), g(x',y') \to^? c$
	$\rightsquigarrow^*$	$g(x',y') \to^? c, x \mapsto x', y \mapsto s(y')$
	$\rightsquigarrow$**Reachability**	**Fail**

Table 5.6: Using constraints to prune AC-goals

Réty, 1991]. For this example, we have used the notion of reachability from [Dershowitz and Sivakumar, 1988].

Constraint	$\langle A \cup \{f(x_1,\ldots,x_n) \to^? t\}; G; \sigma\rangle$
	$\rightsquigarrow$
	Fail
	$f(x_1,\ldots,x_n)\sigma$ does not unify with any left-hand side of R,
	$f(x_1,\ldots,x_n)\sigma$ and t do not unify.

Table 5.7: Transformation rules for constraints

In Example 16 we introduced the notion of constraints. In fact, we can provide a transformation rule for pruning in this case, as shown in Table 5.7.

6 PATH ORDERINGS FOR TERMINATION OF AC-REWRITING

The essential idea in rewriting is to use an asymmetric directed equality ($\rightarrow$), rather than the usual symmetric equality relation ($\approx$). *Termination* of a system consisting of such directed equations means that no infinite sequences of left-to-right replacements are possible for any term. Termination is important for using rewriting as a computational tool, and for simplification in theorem provers. Furthermore, as discussed in Section 2.1, convergence (and therefore termination) is useful in formulating unification procedures (since, with convergence, one can ignore reducible solutions without sacrificing completeness). One popular way of proving termination of a rewrite system is to use *path orderings*, based on a precedence relation on the function symbols of the system. Another common approach interprets function symbols as multivariate polynomials. For a survey of these techniques, see [Dershowitz, 1987].

In this chapter we will consider extended rewriting modulo the axioms of associativity and commutativity. Since it is not possible to orient the commutativity axiom without losing termination, rewriting modulo such a congruence has to be handled in a special way. In essence, we rewrite AC-equivalence classes, rather than terms. AC-functions are very commonplace in practice, thus motivating a need for orderings for proving termination of extended rewriting modulo these axioms.

Polynomials can be used to prove termination of rewriting modulo AC when AC-equivalent terms have the same interpretation. But this severely restricts the degree of polynomial that can be used. (See [Lankford, 1979] and [Ben Cherifa and Lescanne, 1987].) Path orderings have been commonly used in theorem provers, even for AC-rewriting (see the discussion in [Björner, 1982, page 350]), despite the fact that they do not establish termination in the AC case (see the counterexamples in [Dershowitz *et al.*, 1983]). Extensions of path orderings that do handle associative and commutative functions properly ([Bachmair and Plaisted, 1985], for example) have been proposed, most recently in [Kapur *et al.*, 1990]. However, the ordering of [Kapur *et al.*, 1990] is difficult to implement, because it requires many nondeterministic operations (like *pseudocopying*; see Section 6.2).

In this chapter, we show that if a rewrite system can be proved terminating using the recursive path ordering (RPO), then it is also AC-terminating—provided that when comparing two terms with the same (or equivalent) AC symbol at their roots, we compare subterms component wise, rather than as multisets. This criterion can be easily implemented.

6.1 Terminology for AC Systems

We write $s \sim_{ac} t$ to denote that s and t are rearrangements using the AC axioms. AC-rewriting ($\rightarrow_{R/AC}$) can be defined as follows: $u[s]_\pi \rightarrow_{R/AC} u[t]_\pi$, for terms s, t, context u and position π, if $s \sim_{ac} s'$, $s' \rightarrow_R t'$ and $t' \sim_{ac} t$. When dealing with AC systems, it is often convenient to treat AC symbols as functions with variable arity by considering only *flattened* terms. We use $\overline{t}$ to denote the flattened version of t. An ordering $\succ$ is *AC-compatible* if, for all terms s, s', t, t', $s \sim_{ac} s' \succ t' \sim_{ac} t$ implies $s \succ t$, in which case, we can also say that $\overline{s} \succ \overline{t}$. A rewrite system is *AC-terminating* if and only if the relation $\rightarrow_{R/AC}$ is contained in an AC-compatible reduction ordering.

6.2 Binary Path Condition

In this section we develop a restricted version of RPO—called "binary path condition"—which can be extended to an AC-compatible reduction ordering.

We first show that RPO, in general, is not AC-compatible. Consider the rule

$$f(a, f(a, b)) \quad \rightarrow \quad f(b, f(a, a))$$

If we consider $b \succ_f a$, then we can show that $f(a, f(a, b)) \succ_{rpo} f(b, f(a, a))$, assuming multiset status for f. However, we also have that $f(a, f(a, b)) \sim_{ac} f(b, f(a, a))$. Clearly, RPO with lexicographic status is not compatible with the commutativity axiom:

$$f(a, b) \quad \rightarrow \quad f(b, a)$$

If we now have $a \succ_f b$, then using left-to-right status for f, we have $f(a, b) \succ_{rpo} f(b, a) \sim_{ac} f(a, b)$, which violates irreflexivity. Finally, we show that RPO on flattened terms is not AC-

compatible:

$$\begin{aligned} f(a,b) &\rightarrow g(a,b) \\ f(a,g(a,b)) &\rightarrow f(a,a,b) \end{aligned}$$

Here $f \succ_f g$, and f is AC. Now, we have $f(a,a,b) \equiv \overline{f(a,f(a,b))} \succ_{rpo} f(a,g(a,b)) \succ_{rpo} f(a,a,b)$, which violates irreflexivity.

These counterexamples show that RPO with status cannot be extended to an AC-compatible ordering. We therefore define a restricted version of it ($\succ_{bpc}$), which uses RPO with status for the non-AC symbols, but uses RPO without status to compare terms which have equivalent top-level AC operators. Here we use $t \succsim_{bpc} s$ to mean $t \sim_{ac} s$ or $t \succ_{bpc} s$.

Definition 15 (Binary Path Condition). Let $\succ_f$ be a well-founded precedence ordering on the function symbols. We have $t \equiv f(t_1,\ldots,t_n) \succ_{bpc} g(s_1,\ldots,s_m) \equiv s$ if and only if one of the following holds:

1. $t_i \succsim_{bpc} s$ for some i, $1 \leq i \leq n$.
2. $f \succ_f g$, and $t \succ_{bpc} s_j$ for all j, $1 \leq j \leq m$.
3. $f \sim_f g$, f and g are non-AC, and have the same status, and either
   - f has multiset status, and $\{t_1,\ldots,t_n\} \succ_{mul} \{s_1,\ldots,s_m\}$, or,
   - f has lexicographic status, and
     - $(t_1,\ldots,t_n) \succ_{lex} (s_1,\ldots,s_m)$, and
     - $t \succ_{bpc} s_j$ for all j, $1 \leq j \leq m$.
4. $f \sim_f g$, f,g are AC, $t \equiv f(t_1,t_2)$ and $s \equiv g(s_1,s_2)$, and either $(t_1,t_2) \succ_{comp} (s_1,s_2)$ or $(t_1,t_2) \succ_{comp} (s_2,s_1)$, where $(t_1,t_2) \succ_{comp} (s_1,s_2)$ if and only if either $t_1 \succ_{bpc} s_1$ and $t_2 \succsim_{bpc} s_2$, or, $t_1 \succ_{bpc} s_2$ and $t_2 \succsim_{bpc} s_1$.

To compare terms with variables, we can use the fact that a ground term $t\sigma$ is greater under $\succ_{bpc}$ than $x\sigma$ (x is a variable), for any substitution σ, whenever x occurs in t.

Theorem 20. *Let R be a rewrite system. If for each rule $l \rightarrow r \in R$ we have $l \succ_{bpc} r$, then R is AC-terminating.*

We first recall the definition of AC-RPO ($\succ_{ac}$), on ground terms, due to Kapur *et al.*, [1990]. This ordering compares flattened terms.

Definition 16. Let $\succ_f$ be a well-founded precedence ordering on the function symbols. We have $t \equiv f(t_1, \ldots, t_n) \succ_{ac} g(s_1, \ldots, s_m) \equiv s$ if and only if one of the following holds:

1. $t_i \succsim_{ac} s$ for some i, $1 \leq i \leq n$, where $t_i \succsim_{ac} s$ if and only if $t_i \sim_{ac} s$ or $t_i \succ_{ac} s$.

2. $f \succ_f g$, and $t \succ_{ac} s_j$ for all j, $1 \leq j \leq m$.

3. $f \sim_f g$, f and g are non-AC and have the same status, and either

   - f has multiset status, and $\{t_1, \ldots, t_n\} \succ_{mul} \{s_1, \ldots, s_m\}$, or,
   - f has lexicographic status, and
     - $(t_1, \ldots, t_n) \succ_{lex} (s_1, \ldots, s_m)$, and
     - $t \succ_{ac} s_j$ for all j, $1 \leq j \leq m$.

4. $f \sim_f g$, f, g are AC, $t = f(T)$, $s = g(S)$, $S' = S - T = \{s'_1, ..., s'_k\}$ (where "$-$" denotes the multiset difference performed using $\sim_{ac}$, i.e., terms equivalent with respect to $\sim_{ac}$ can be dropped from both T and S), and either

   - $k = 0$ and $n > m$ (i.e., $S - T = \emptyset$ and $T - S \neq \emptyset$), or
   - $f(T - S) \Rightarrow^* f(T')$, and $T' = T_1 \cup \ldots \cup T_k$ and for all i $(1 \leq i \leq k)$ either
     - $T_i = \{u\}$ and $u \succsim_{ac} s'_i$, or
     - $T_i = \{u_1, \ldots, u_l\}$ and $f(u_1, \ldots, u_l) \succsim_{ac} s'_i$.

     Also, in this case, either $t \Rightarrow^+ f(T')$, or, for at least one i, we have instead a strict decrease in $\succ_{ac}$.

Case 4 of this definition uses the operation $\Rightarrow$, which may be one of the following: *pseudo-copying*, *elevation* or *flattening*. Here we briefly explain these notions; for details refer to [Kapur *et al.*, 1990]. Pseudo-copying is used to allow a single *big* (that is, with a top-level function which is higher than f in the precedence relation $\succ_f$) subterm on the left-hand side to handle multiple subterms on the right. For example, while comparing the terms $t_1 \equiv f(g(x))$ and $t_2 \equiv f(h(x), h(x))$, where f is AC, and $g \succ_f f \succ_f h$, we can say that $t_1 \succ_{ac} t_2$, since

$t_1 \Rightarrow f(gg(x), gg(x)) \succ_{ac} t_2$, where $gg(x)$ is a pseudo-copy of $g(x)$. Note that pseudo-copying is allowed only for big terms which are immediate subterms of the top-level AC operator of the left-hand side term. At times, a big term may be nested further down, in which case elevation is used to bring it up. For example, in comparing $f(c(g(x)))$ with $f(h(x), h(x))$, where $g \succ_f f \succ_f h \succ_f c$, we can use the following steps: $f(c(g(x))) \Rightarrow f(g(x)) \succ_{ac} f(h(x), h(x))$. Finally, flattening can be used to remove immediate nesting of different AC functions which have the same precedence. For example, we could say $f(g(x), y) \Rightarrow f(x, y)$, if $f \sim_f g$, and f and g are AC. The essential idea in this ordering is to partition the subterms of the AC functions and compare the components, using $\Rightarrow$ to make the relation transitive. It is shown in [Kapur *et al.*, 1990] that this ordering is well-founded and AC-compatible.

We are ready for a proof of the theorem:

Proof. We show that for any two terms s and t, if $t \succ_{bpc} s$ then $\overline{t} \succ_{ac} \overline{s}$, by induction on the sizes of s and t. There are several cases to be considered, depending on the possible reasons why $t \succ_{bpc} s$. We assume that t is of the form $f(t_1, \ldots, t_n)$ and s is $g(s_1, \ldots, s_m)$.

1. If $t_i \succsim_{bpc} s$, then, by the inductive hypothesis we have $t_i \succsim_{ac} s$, and hence $\overline{t} \succ_{ac} \overline{s}$, by Case 1 of Definition 16.

2. If $f \succ_f g$ and $t \succ_{bpc} s_j$, then by the inductive hypothesis, we have $\overline{t} \succ_{ac} \overline{s_j}, 1 \leq j \leq m$. There are two further possibilities:

   - g is not AC. In this case, $\overline{s} \equiv g(\overline{s_1}, \ldots, \overline{s_m})$, and therefore we have $\overline{t} \succ_{ac} \overline{s}$, by Case 2 of Definition 16.
   - Suppose g is AC (thus $m = 2$). In this case, there are various possibilities for s, for example we could have: $s \equiv g(g(s_{1.1}, s_{1.2}), s_2)$, or $s \equiv g(s_1, g(s_{2.1}, s_{2.2}))$, or $s \equiv g(g(s_{1.1}, s_{1.2}), g(s_{2.1}, s_{2.2}))$, and so forth. However, in each case, we have $\overline{s} \equiv g(s'_1, \ldots, s'_k)$, where each $s'_j, 1 \leq j \leq k$, is either a subterm of s_1 or of s_2. Therefore, we have $\overline{t} \succ_{ac} \overline{s}$.

3. If f and g are both non-AC, and $f \sim_f g$, then we can use the inductive hypothesis on the flatten subterms of s, and then the proposition follows.

4. If f and g are both AC, and $f \sim_f g$ then $n = m = 2$. Furthermore, without loss of generality, we can assume that $t_1 \succ_{bpc} s_1$, *and* $t_2 \succsim_{bpc} s_2$. (The other cases admit similar proofs.) There are two further possibilities:

   - If $t_2 \succ_{bpc} s_2$, then by the inductive hypothesis we have: $\overline{t_1} \succ_{ac} \overline{s_1}$ and $\overline{t_2} \succ_{ac} \overline{s_2}$. Thus, we could use this partitioning of t to show that $\overline{t} \succ_{ac} \overline{s}$.
   - If $t_2 \sim_{ac} s_2$, then again the proposition holds, because we could ignore $\overline{t_2}$ and $\overline{s_2}$ when comparing $\overline{t}$ with $\overline{s}$.

□

Since $\succ_{ac}$ is AC-compatible, we have $\overline{t} \succ_{ac} \overline{s}$, not only when $t \succ_{bpc} s$, but also if $t \sim_{ac} t' \succ_{bpc} s$. In order to prove termination of a system using $\succ_{bpc}$, it is therefore sufficient to use any rearrangement of the left- and right-hand side terms. We also have, for any AC function symbol f and terms s and t, that if $t \succ_{bpc} s$, then $f(t, X) \succ_{bpc} f(s, X)$ (and therefore, $\overline{f(t, X)} \succ_{ac} \overline{f(s, X)}$).

We have shown that the relation $\succ_{bpc}$ is embedded in the AC-compatible reduction ordering $\succ_{ac}$. Therefore, the binary path condition is sufficient for proving AC-termination. It is important to note that the relation ($\succ_{bpc}$) defined here is not really an ordering, because it is not transitive. For example, if we have the precedence relation $g \succ_f f \succ_f h$, then we can show that (here f is AC, while g and h are non-AC)

$$f(g(x), g(y)) \succ_{bpc} f(f(x,x), f(y,y)) \sim_{ac} f(f(x,y), f(x,y)) \succ_{bpc} f(h(x,y), h(x,y)).$$

However, it is the case that $f(g(x), g(y)) \not\succ_{bpc} f(h(x,y), h(x,y))$. The interesting point about $\succ_{bpc}$ is that it is easy to implement; much easier than the ordering of [Kapur *et al.*, 1990].

6.3 Examples

The binary path condition developed in the previous section, like $\succ_{ac}$, and unlike [Bachmair and Plaisted, 1985], has no restriction on the precedence relation $\succ_f$, and can therefore be used to prove termination of a large class of rewrite systems.

Example 17 (Arithmetic over natural numbers). Here $*$ and $+$ are AC, and $* \succ_f + \succ_f s \succ_f 0$.

$$\begin{aligned} 0 + x &\rightarrow x \\ s(x) + y &\rightarrow s(x + y) \\ 0 * x &\rightarrow 0 \\ s(x) * y &\rightarrow y + (x * y) \\ (x + y) * z &\rightarrow (x * z) + (y * z) \end{aligned}$$

Example 18 (Free commutative ring). Here $*$ and $+$ are AC, and $* \succ_f - \succ_f + \succ_f 0$.

$$\begin{aligned} 0 + x &\rightarrow x \\ -x + x &\rightarrow 0 \\ -0 &\rightarrow 0 \\ - - x &\rightarrow x \\ -(x + y) &\rightarrow -x + -y \\ 0 * x &\rightarrow 0 \\ -x * y &\rightarrow -(x * y) \\ x * (y + z) &\rightarrow (x * y) + (x * z) \end{aligned}$$

6.4 Discussion

In this chapter, we have considered a simple restriction on RPO that can be extended to an AC-compatible ordering. The restriction disallows comparison of two terms with equivalent top-level AC functions when both subterms on the right-hand side are dominated by only one subterm on the left-hand side.

Independently, Bachmair [1992] presented an AC-compatible rewrite-relation, also based on [Kapur *et al.*, 1990], and proved its termination using a minimal counterexample argument. Our termination condition is essentially the same as one rewrite step of [Bachmair, 1992], with the possibility of multiset status added. It is believed that the transitive closure of our relation is identical to the ordering in [Kapur *et al.*, 1990], but this remains to be proved.

It will be interesting to be able to extend the relation defined here to cases where simple multiset comparisons may be allowed for subterms of the AC-terms. However, as shown

in [Bachmair, 1992], even simple rules like $f(g(x,y),z) \rightarrow f(x,y)$, may lead to non-terminating rewrite sequences.

The binary path condition as a sufficient condition for AC-termination was presented in [Dershowitz and Mitra, 1992]. More recently, Rubio and Nieuwenhuis [1993] have extended the ordering of [Kapur *et al.*, 1990] to one that is total on (non-AC equivalent) ground terms. However, one inconvenience with their ordering is that it orients the distributivity axiom (for example, $x * (y + z) = x * y + x * z$) the "wrong" way. We believe that a lexicographic combination of BPC (or, for that matter, any other ordering, for example, [Kapur *et al.*, 1990; Bachmair, 1992; Delor and Puel, 1993], that orients distributivity the right way) together with the ordering of [Rubio and Nieuwenhuis, 1993] would solve the problem.

7 DECIDABLE EQUATION SOLVING

The transformation systems described in Chapter 3 are complete for unification and matching in convergent rewrite systems. However, it is well-known that both these problems are, in general, undecidable [Bockmayr, 1987; Heilbrunner and Hölldobler, 1987; Dershowitz and Jouannaud, 1990]. Hullot [1980], Kapur and Narendran [1987] and Christian [1992] have provided different syntactic restrictions on the rewrite system which result in decidable unification in theories defined by convergent systems. However, these restrictions are quite strong from a practical point of view. In this chapter, we approach the problem in two parts, and therefore provide a better characterization for decidability. We first use the restricted sets of transformation rules from Section 3.5 to provide syntactic and semantic requirements on the rewrite system which result in decidable matching. Thereafter, we address the general problem, and propose a characterization of systems with *flat* right-hand sides, which results in decidable unification. Roughly, flat systems disallow nested function symbols in the right-hand sides, and allow variables on the right-hand sides to appear under at most one function symbol.

In this chapter, we say that a function symbol f is a *defined function* if it appears at the root of some left-hand side of a rule in a rewrite system R; if there is no such rule, then f is a *constructor*.

7.1 Undecidable Matching and Unification Problems

Throughout this chapter, we will use some well-known problems to show that certain new problems are undecidable. In this section we list two such undecidable problems:

Example 19. The following convergent system has an undecidable semantic unification problem:

$$\begin{aligned} 1 + x &\rightarrow s(x) \\ s(x) + y &\rightarrow s(x + y) \\ \\ 1 * x &\rightarrow x \\ s(x) * 1 &\rightarrow s(x) \\ s(x) * s(y) &\rightarrow s(y + (x * s(y))) \end{aligned}$$

The system defines addition ($+$) and multiplication ($*$) over positive integers; integers being represented in unary notation, using the constant 1 and successor function s.

It can be shown that, in general, it is undecidable if an equation has a solution with respect to the rewrite system given above (since were there a decision procedure for this, it would solve Hilbert's undecidable Tenth Problem). We will prove later that the semantic matching problem is, nevertheless, decidable for this theory. ([Bockmayr, 1987; Dershowitz and Jouannaud, 1990] use similar examples to show that, in general, semantic matching and unification are undecidable for convergent systems.)

We will also use the following variant of the above example, which has similar properties (that is, the matching problem is decidable, while the unification problem is not):

Example 20.

$$\begin{aligned} 0 + x &\rightarrow x \\ s(x) + y &\rightarrow s(x + y) \\ 0 * x &\rightarrow 0 \\ x * 0 &\rightarrow 0 \\ s(x) * s(y) &\rightarrow s(y + (x * s(y))) \end{aligned}$$

The next example that we use is from [Heilbrunner and Hölldobler, 1987], wherein a scheme to construct a convergent system of rewrite rules has been discussed, given two (simple) context-free grammars in Greibach normal form. Here we provide an example (see [Heilbrunner and Hölldobler, 1987] for details about termination and confluence of the resulting rewrite system):

Example 21. Let G_1 and G_2 be two context free grammars, as described below:

$$G_1 \equiv \{S_1 \Rightarrow \mathrm{aB}_1C_1, B_1 \Rightarrow b, C_1 \Rightarrow c\}$$
$$G_2 \equiv \{S_2 \Rightarrow \mathrm{aB}_2, B_2 \Rightarrow \mathrm{bC}_2, B_2 \Rightarrow a, C_2 \Rightarrow c\}$$

Here S_1 and S_2 are the *start symbols*; B_1, B_2, C_1 and C_2 are the *non-terminals*, while a, b and c are the *terminal-symbols*. Consider the rewrite system R:

$$\begin{aligned} f(S_1 \cdot x, B_2 \cdot y, a \cdot z) &\rightarrow f(B_1 \cdot (C_1 \cdot x), y, z) \\ f(S_1 \cdot x, S_2 \cdot y, a \cdot z) &\rightarrow f(B_1 \cdot (C_1 \cdot x), B_2 \cdot y, z) \\ f(B_1 \cdot x, B_2 \cdot y, b \cdot z) &\rightarrow f(x, C_2 \cdot y, z) \\ f(C_1 \cdot x, C_2 \cdot y, c \cdot z) &\rightarrow f(x, y, z) \end{aligned}$$

For simplicity, we have used the same signature as the grammars given above. Thus, for R, f is the only defined function, while all the other functions are constructors.

If can be shown that the function f, defined by R, has the following property:

$$f(S_1 \cdot \$, S_2 \cdot \$, x) \stackrel{?}{=} f(\$, \$, \$) \text{ if and only if } x \in (G_1 \cap G_2),$$

where \$ is the end symbol.

Heilbrunner and Hölldobler [1987] show how f can be constructed given any two arbitrary context-free grammars G_1 and G_2 in simple Greibach normal form. Therefore, in general, there can be no decision procedure to solve the problem $f(S_1 \cdot \$, S_2 \cdot \$, x) \stackrel{?}{=} f(\$, \$, \$)$ (since such a decision procedure would also decide if the intersection of two simple context-free languages is empty, which is undecidable [Heilbrunner and Hölldobler, 1987]).

7.2 Decidable Matching

As noted earlier, in the most general case, semantic matching can be as difficult as full semantic unification: For example, adding a new rule $eq(x, x) \rightarrow \mathit{true}$ to the system of Example 19 makes the problem of unifying two terms s and t the same as matching $eq(s, t)$ to *true* in the augmented theory. In this section we use the syntactic characterizations given in Section 3.5 (those of non-erasing and left-linear systems) to show that the matching problem is decidable for special classes of such systems.

7.2.1 Non-Erasing Rules

In looking for decidable matching problems, we started with the following result (a special case of Theorem 22 which we prove later):
If R is a non-erasing convergent term rewriting system for which:

- *all right-hand sides of rules are either variables, or have a constructor at the top-level, and*
- *there are no nested defined functions in any right-hand side,*

then the semantic matching problem is decidable for R.

Example 22. By the above result, the following system has a decidable semantic matching problem.

$$\begin{aligned} app(nil, x) &\rightarrow x \\ app(x \cdot y, z) &\rightarrow x \cdot app(y, z) \end{aligned}$$

In [Heilbrunner and Hölldobler, 1987], there is an example of a system with a single defined function in every right-hand side, which has an undecidable semantic matching problem (Example 21 shows the construction for a particular case). There, the defined function on the right-hand side of rules does not appear below a constructor, but it obeys the other restrictions. This shows that defined functions must appear below at least one constructor.

Next, we tried to allow some nested defined functions on the right-hand sides of the rewrite rules. We require the following definitions:

Definition 17 (Suitable Property). A *suitable property* is a measure $\mathcal{P}$ (like *depth*, *size*, etc.) associated with ground terms, along with a well-founded total ordering $>$ that compares values of $\mathcal{P}$, such that $\mathcal{P}$ is strictly larger, under $>$, for terms than for its subterms.

Definition 18 (Non-Decreasing). A function symbol f is defined to be *non-decreasing* (with respect to a suitable property $\mathcal{P}$) if whenever $f(\widehat{s_1}, \ldots, \widehat{s_n}) \rightarrow^! N$, where each $\widehat{s_i}$ and N is in ground normal form, $\mathcal{P}(\widehat{s_i}) \leq \mathcal{P}(N)$. Any function which does not have this property is said to be a *potentially decreasing* function (with respect to $\mathcal{P}$).

Similarly, we can define the notion of "strict increasingness" for a function.

Unfortunately, it is not possible to always decide whether a function defined by a given convergent rewrite system is non-decreasing with respect to a property $\mathcal{P}$, even for a simple suitable property like depth:

Lemma 21. *It is undecidable if a function symbol is depth non-decreasing.*

Proof. Consider the system:

$$\begin{aligned} g(x) &\rightarrow h(f(S_1 \cdot \$, S_2 \cdot \$, x), x) \\ h(f(\$, \$, \$), x) &\rightarrow \$ \end{aligned}$$

where f is as detailed in [Heilbrunner and Hölldobler, 1987] (Example 21 shows the construction of f for a particular set of context-free grammars). Assuming convergence of f, it is easy to show that this combined system is also convergent.

If S_1 and S_2 are respectively the start symbols for two context free grammars G_1 and G_2, we have

$$f(S_1 \cdot \$, S_2 \cdot \$, x) \rightarrow^! f(\$, \$, \$) \textit{ if and only if } x \in G_1 \textit{ and } x \in G_2.$$

By the above construction, g can be depth non-decreasing if and only if

$$\forall x.x \notin (G_1 \cap G_2).$$

Thus, a decision procedure for this problem could be used to decide if the intersection of two arbitrary context free grammars is empty, which is impossible. □

On the brighter side, certain decidable subclasses of functions that are non-decreasing (increasing) are easy to identify. For example, any function which has a variable dropping rule, with the dropped variable appearing immediately below the top-level function on the left-hand side, cannot be depth non-decreasing (increasing). Again, for each rule $l \rightarrow r$ which defines a function f, if $depth(l) \leq depth(r)$ then f is depth non-decreasing. We can also have similar sufficient conditions using the depth of each variable in the rule. For example, if every variable occurs below at least the same number of constructors on the right-hand side, as on the left-, then the corresponding function is depth non-decreasing. We can use the last criterion to show that $+$, as defined in Example 19, is depth non-decreasing.

Unfortunately, if the right-hand sides in rewrite rules have defined functions nested below a potentially (depth) decreasing function, then the resulting system may have undecidable semantic matching problems:

Example 23. Consider the rules below, together with the definitions of $+$ and $*$ given in Example 19 (the combined system is convergent):

$$\begin{aligned} half(s(1)) &\rightarrow 1 \\ half(s(s(x)) &\rightarrow s(half(x)) \end{aligned}$$

$$\begin{aligned} f(1,1) &\rightarrow s(1) \\ f(s(x),s(y)) &\rightarrow s(half(f(x,y))) \end{aligned}$$

Here *half* is a potentially (depth) decreasing function. We have the following property for f:

$$f(x,y) = \begin{cases} s(1) & \text{if } x = y = s^n(1), n \geq 0 \\ undefined & \text{otherwise} \end{cases}$$

(Here, we have used the notation $s^n(1)$ as a short-hand: $s^0(1) = 1$ and $s^{i+1}(1) = s(s^i(1)), i \geq 0$.) We can now try to solve the goal $f(t_1, t_2) \rightarrow^? s(1)$, where t_1 and t_2 are terms involving $+$ and $*$. This goal has a solution σ if and only if $t_1\sigma$ and $t_2\sigma$ have the same ground normal form (because of the observation made about f before). Thus, if this problem has a decision procedure, then we could use the same for deciding the semantic unification problem mentioned in Example 19. Therefore, no such decision procedure can exist.

Based on the counterexample above, it can be seen that a function definition in terms of some potentially decreasing functions is not suitable for our purpose. We, therefore, restrict the right-hand sides of rules to only have potentially decreasing functions at the lowest level (that is, no other defined function symbol can be nested below them). We have:

Theorem 22. *Let R be a convergent non-erasing term rewriting system, and $\mathcal{P}$ be some suitable property. If*

- *all right-hand sides for rules in R are either variables, or have a constructor at the top-level, and*

- *all right-hand sides are such that no defined function is nested below any function decreasing with respect to $\mathcal{P}$,*

then the semantic matching problem is decidable for R.

Proof. Let $\succ$ be a well-founded ordering on goals such that $s_1 \rightarrow^? N_1 \succ s_2 \rightarrow^? N_2$ if and only if either $\mathcal{P}(N_1) > \mathcal{P}(N_2)$ or $\mathcal{P}(N_1) = \mathcal{P}(N_2)$ and s_2 is a subterm of s_1.

We show that it is possible to find all solutions (in finite time) to any goal of the form $\varrho \rightarrow^? N$, where ϱ is a term which has no defined function nested below any decreasing function and N is a ground normal form. This we do by induction with respect to the ordering $\succ$.

The interesting case is the one in which $\varrho \equiv f(\varrho_1, \ldots, \varrho_n)$, and f is a defined function. It is, therefore, possible to use the transformation rule Mutate on this goal, applying some rule $f(l_1, \ldots, l_n) \rightarrow \rho$. The essential steps are:

$$\begin{array}{lll} \{\varrho \rightarrow^? N\} & \rightsquigarrow_{\textbf{Mutate}} & \{\varrho_i \rightarrow^? l_i, \ldots, \varrho_n \rightarrow^? l_n, \rho \rightarrow^? N\} \\ & \rightsquigarrow_{\textbf{Decompose}} & \{\varrho_i \rightarrow^? l_i, \ldots, \varrho_n \rightarrow^? l_n, \rho_1 \rightarrow^? N_1, \ldots, \rho_m \rightarrow^? N_m\} \end{array}$$

Since every right-hand side of a renamed rule in R, by assumption, has constructors at the top, at least one application of Decompose is possible, starting with the goal $\rho \rightarrow^? N$. In the derivation above, we have shown such a decomposition step, assuming that the top-level constructor of ρ has m immediate subterms.

The subgoals $\{\rho_j \rightarrow^? N_j\}$ produced after the decomposition step are smaller than the original goal, that is, $\{\varrho \rightarrow^? N\} \succ \{\rho_j \rightarrow^? N_j\}$, for each j. Thus, by applying the inductive hypothesis we can assume that all the solutions to each of the goals in $\{\rho_j \rightarrow^? N_j\}$ (and therefore also for their collection, that is, $\rho \rightarrow^? N$) can be found in finite time. Let σ be the solution obtained along one feasible branch for the goal $\rho \rightarrow^? N$. Since all rules are non-erasing, ρ contains all the variables that appear in any of the l_i subterms. Furthermore, because of the non-erasing nature of all rules, σ must be a ground solution (if not, we will have a situation where a non-ground term will rewrite using only non-erasing rules to a ground term, which is not possible). Thus, for any such σ, each $l_i\sigma$ must be ground. (See the proof of Theorem 8, which discusses a selection strategy for picking subgoals to be solved; we continue to use the same selection strategy for solving goals in this proof.)

There are now two different cases to be considered.

- Function f is *potentially decreasing*. By assumption there is no defined function below it, that is, no ϱ_i has a defined function, and therefore all the $\varrho_i \rightarrow^? l_i$ subgoals can be decomposed immediately to solved forms ($x \mapsto N'$), or to unsolvable goals with different constructors at the top. Therefore, all solutions for $\varrho \rightarrow^? N$ can be found in finite time in this case.

- Function f is *non-decreasing*. The important point to note is that each left-hand side in the list of subgoals (that is, $\varrho_i, 1 \leq i \leq n$) has the property that no defined function is nested below a potentially decreasing function. Let us now consider the ground solution σ (for $\rho \rightarrow^? N$) as described above. Since f is known to be non-decreasing with respect to $\mathcal{P}$, each of the $l_i\sigma$ terms must be such that $\mathcal{P}(l_i\sigma) \leq \mathcal{P}(N)$, or else the partial solution σ violates the condition that f is non-decreasing, and can be ignored. (In this case, the goal $\varrho \rightarrow^? N$ has no solution, using the rule $f(l_1, \ldots, l_n) \rightarrow \rho$, for Mutate.)

  Thus, for all feasible paths, we have that $\mathcal{P}(l_i\sigma) \leq \mathcal{P}(N)$, and therefore, we get

$$\{\varrho \stackrel{?}{\rightarrow} N\} \succ \{\varrho_i \stackrel{?}{\rightarrow} l_i\sigma\},$$

  for each i, since each ϱ_i is a subterm of ϱ. Using the induction hypothesis, each subgoal $\varrho_i \rightarrow^? l_i\sigma$ can be solved, and therefore the goal $\varrho \rightarrow^? N$ itself can also be solved.

Using this result, it is easy to show (by induction on the size of the left-hand sides of goals) that for any term s (even without the restrictions imposed on ϱ), and ground normal form $\widehat{N}$, the goal $s \rightarrow^? \widehat{N}$ is solvable. The idea is that for every application of Mutate with the goal $s \rightarrow^? \widehat{N}$, the subgoal $r \rightarrow^? N$ is solvable (by the above argument). Thus, we can replace the multiset of subgoals generated by Mutate, by a finite number of such multisets (each without $r \rightarrow^? N$) corresponding to each of the solutions of $r \rightarrow^? N$. □

In certain special cases it is possible to relax the requirement that all right-hand sides with defined functions must have a constructor at the top-level. For example, if we assume that the top-level function on the right-hand side is strictly increasing, and that it eventually generates a constructor in a finite number of steps, then the above theorem would still hold for such systems. The following example illustrates the point:

Example 24.

$$\begin{aligned} 1 + x &\rightarrow s(x) \\ s(x) + y &\rightarrow s(x + y) \end{aligned}$$

$$\begin{aligned} fib(1) &\rightarrow 1 \\ fib(s(1)) &\rightarrow 1 \\ fib(s(s(x))) &\rightarrow fib(s(x)) + fib(x) \end{aligned}$$

Here, $+$ is a strictly (depth) increasing function, and fib defines the Fibonacci numbers, both being defined over positive integers. Furthermore, both rules for $+$ have constructors at the top-level on the right-hand side, and the remaining rules have the properties required by Theorem 22. Therefore, the semantic matching problem is decidable for this system.

The essential idea is that if any sequence of applications of Mutate for increasing functions generate a constructor eventually, then the matching problem is decidable. Here is an outline of the proof:

If the top-level symbol (of ϱ in Theorem 22) is an increasing function, the applicable transformation rule generates the following derivation:

$$\{\varrho \rightarrow^? N\} \quad \rightsquigarrow_{\mathbf{Mutate}} \quad \{\varrho_{i_1} \rightarrow^? l_{i_1}, \rho_1 \rightarrow^? N\}$$

The goal $\rho \rightarrow^? N$ is not decreasing as such. However, since we assumed that all such derivations eventually generate a constructor at the top, we must have at least one step of decomposition if we continue to mutate this goal. The derivation therefore would look like:

$$\begin{aligned} \{\varrho \rightarrow^? N\} \quad &\rightsquigarrow_{\mathbf{Mutate}^*} \quad \{\varrho_{i_1} \rightarrow^? l_{i_1}, \ldots, \varrho_{i_m} \rightarrow^? l_{i_m}, \rho_m \rightarrow^? N\} \\ &\rightsquigarrow_{\mathbf{Decompose}} \quad \{\varrho_{i_1} \rightarrow^? l_{i_1}, \ldots, \varrho_{i_m} \rightarrow^? l_{i_m}, \bar{\rho_m} \rightarrow^? \bar{N}\} \end{aligned}$$

We assume that we only mutate the $\rho_m \rightarrow^? N$ subgoal at every stage, in keeping with the selection strategy mentioned in the completeness proof of Section 3.5. Now, we can show that the subgoals are decreasing with respect to the ordering $\succ$, as in Theorem 22.

Notice that the system described in Example 19 obeys all the restrictions of Theorem 22, and thus has a decidable semantic matching problem.

7.2.2 Erasing Rules

In this section we deal with the possibility of incorporating erasing rules into the rewrite system, and will try to extend Theorem 22 suitably to handle such cases. However, before we do so, we point out some cases which cause problems, by way of counterexamples.

Example 25. Consider the rule given below, together with the definitions of $+$ and $*$ from Example 19

$$eq(x, x) \quad \rightarrow \quad \text{true}$$

This is the only erasing rule in the system. For this set of rules, the goal $\{eq(s,t) \rightarrow^? true\}$ is not solvable in general, because, once again, a solution to this problem would mean a decision procedure for some variation of the Hilbert's Tenth Problem.

Example 26. This time we consider two rules and the definitions of $+$ and $*$ as before

$$\begin{aligned} f(s(x), 0) &\rightarrow 0 \\ eq(x, x) &\rightarrow x \end{aligned}$$

Here, the only erasing rule is the one for f. Consider a goal of the form $\{f(eq(t_1, t_2), y) \rightarrow^? 0\}$, where t_1 and t_2 are terms involving $+$ and $*$. The possible derivation steps for this goal are shown below:

$$\begin{aligned} \{f(eq(t_1, t_2), y) \rightarrow^? 0\} &\rightsquigarrow_{\mathbf{Mutate}} \{eq(t_1, t_2) \rightarrow^? s(x_1), y \mapsto 0\} \\ &\rightsquigarrow_{\mathbf{Mutate}} \{t_1 \rightarrow^? x, t_2 \rightarrow^? x, y \mapsto 0, x \mapsto s(x_1)\} \end{aligned}$$

Thus, if this goal is solvable, then we can also solve the unification problem with respect to $+$ and $*$, which leads to a contradiction, since the latter problem is known to be undecidable. This example illustrates the fact that a system with a single (left-linear) erasing rule may admit undecidable matching problems, even when the other rules are non-erasing.

It is important to note that a single erased variable may interact with non-linear variables of the left-hand side of some other rule in a way which may lead to undecidability. Here we give an example which has a single non left-linear rule, and an erasing rule.

Example 27. Consider the following rules, together with the definitions of $+$ and $*$ as before:

$$\begin{aligned} eq(x,x,y) &\rightarrow s(g(x,y)) \\ g(x,0) &\rightarrow true \end{aligned}$$

With this system of rules, and the goal $eq(t_1,t_2,z) \rightarrow^? s(true)$, where t_1 and t_2 are terms involving $+$ and $*$, we have the following derivations:

$$\begin{aligned} \{eq(t_1,t_2,z) \rightarrow^? s(true)\} &\leadsto_{\mathbf{Mutate}} \{t_1 \rightarrow^? x, t_2 \rightarrow^? x, z \rightarrow^? y, s(g(x,y)) \rightarrow^? s(true)\} \\ &\leadsto^* \{t_1 \rightarrow^? x, t_2 \rightarrow^? x, y \mapsto 0, z \mapsto 0, x \mapsto x_1, \ldots,\} \end{aligned}$$

Thus, for reasons similar to Example 26, this problem is undecidable.

The last example shows that even a linear variable occurring immediately below the top-level symbol on the left-hand side of a rule may result in the elimination of a variable that is non-linear in another rule. There is a subtle difference between Examples 26 and 27. In the former, the dropped variable x appears below some constructor (in the first rule), which implies that the subgoal $eq(t_1,t_2) \rightarrow^? s(x_1)$ has to be mutated further, thereby causing the problem. Note that if this rule had a variable (say z) instead of the $s(x)$ term, then the corresponding subgoal $(eq(t_1,t_2) \rightarrow^? z)$ would have been trivially solvable. The last example is a variation of Example 25; the only difference being, in this case, eq drops variables one at a time.

It is possible to have erasing rules and still have a decidable semantic matching algorithm:

Theorem 23. *Let R be a left-linear rewrite system and $\mathcal{P}$ be a suitable property. If*

- *all right-hand sides of rules are either variables, or have a constructor at the top-level, and*
- *there are no nested defined functions in any right-hand sides,*

then the semantic matching problem is decidable for R.

Proof. Since we now have a left-linear system, we only need to solve for goals of the form $s \rightarrow^? t$, where any variable x in t has the property that it is linear in t and does not occur in

the right-hand side of any other subgoal. Thus, we can apply a proof quite similar to that of Theorem 22, and show that the complete matching procedure is terminating.

We use a well-founded ordering (like $\succ$ in Theorem 22), which compares goals using a suitable property for right-hand sides and subterm property for the left-hand sides of goals. Let us consider a goal of the form $\varrho \to^? N(\bar{x})$, with ϱ is a term without any nested defined functions. We use the notation $N(\bar{x})$ to denote a term which has some variables $\bar{x}$, such that N is linear with respect to each of them; furthermore, no other subgoal in the multiset of goals being solved has any of these variables in a right-hand side. We show by induction that all solutions to such goals can be finitely generated. As before, consider an application of Mutate, with the rule $f(l_1, \ldots, l_n) \to \rho$ (assume that $\varrho \equiv f(\varrho_1, \ldots, \varrho_n)$). We have the following derivation (like in Theorem 22):

$$\begin{array}{lll} \{\varrho \to^? N(\bar{x})\} & \rightsquigarrow_{\mathbf{Mutate}} & \{\varrho_i \to^? l_i, \rho \to^? N(\bar{x})\}, 1 \leq i \leq n \\ & \rightsquigarrow_{\mathbf{Decompose}} & \{\varrho_i \to^? l_i, \rho_j \to^? N(\bar{x})_j\} \end{array}$$

We can now apply the inductive hypothesis on the $\rho_j \to^? N(\bar{x})_j$ subgoals, which implies that $\rho \to^? N(\bar{x})$ itself is solvable. Let σ be a solution to this subgoal. By assumption, there are no nested defined functions in ϱ. Therefore, each of the remaining goals can be solved using decomposition alone. □

We next attempt to introduce nested defined functions on the right-hand sides of rules. The main difficulty is with the ordering using a suitable property. For the general case, like in Theorem 22, we have to show that each of the $l_i\sigma$ terms for goals of the form $\varrho_i \to^? l_i\sigma$ are smaller than the original goal, which may not be possible if the right-hand sides of goals contain variables. Thus, further restrictions are required: We restrict the system so that if ρ is the right-hand side of a rule which has a function that has erasing rules, then we require that all goals of the form $\rho \to^? N$ have only ground solutions. If this condition is satisfied, then we can assume that all solutions to the goal $\rho \to^? N(\bar{x})$ is ground, and the proof of Theorem 22 is still valid in this situation. (In effect, we are trying to combine Theorems 22 and 23.) Here we provide examples of two systems which have the required property:

Example 28. In this example $+$ is depth non-decreasing, while $*$ is potentially depth decreasing, and has erasing rules. Also, right-hand sides of rules for $+$ only uses the same function

recursively (which is acceptable, since $+$ does not have erasing rules). Furthermore, for $*$, the last rule (which is the only one which has the function $*$ on the right-hand side) satisfies the condition mentioned above (that is, $y + (x * s(y)) \rightarrow^? N$ admit ground solutions alone, because of properties of multiplication).

$$\begin{aligned} 0 + x &\rightarrow x \\ s(x) + y &\rightarrow s(x + y) \\ \\ 0 * x &\rightarrow 0 \\ s(x) * 0 &\rightarrow 0 \\ s(x) * s(y) &\rightarrow s(y + (x * s(y))) \end{aligned}$$

In the next example, *insert* is a strictly depth increasing function, which uses *min* (a variable erasing function) in its right-hand side (the last rule). However, since both the variables which can potentially be dropped (x and y of the last rule) also appear in the second subterm of the root, under a non-variable dropping function *max*, the entire rule can be treated as non-erasing.

Example 29.

$$\begin{aligned} min(x, 0) &\rightarrow 0 \\ min(0, x) &\rightarrow 0 \\ min(s(x), s(y)) &\rightarrow s(min(x, y)) \\ \\ max(x, 0) &\rightarrow x \\ max(0, x) &\rightarrow x \\ max(s(x), s(y)) &\rightarrow s(max(x, y)) \\ \\ sort(nil) &\rightarrow nil \\ sort(x \cdot y) &\rightarrow insert(x, sort(y)) \\ \\ insert(x, nil) &\rightarrow x \cdot nil \\ insert(x, y \cdot z) &\rightarrow min(x, y) \cdot insert(max(x, y), z) \end{aligned}$$

Thus, in order to introduce nested defined functions on the right-hand sides, we have to ensure that whenever there is a possibility of a variable being dropped, there must be another subgoal

which instantiates that variable. We will continue our discussion on decidable matching in theories with erasing rules in Section 7.2.4.

7.2.3 Matching with Restricted Goals

In this section we will consider matching problems in which the left-hand side of the initial goal is restricted, very much like the right-hand sides of rules of the convergent rewrite system under consideration. For the remainder of this section, we will uniformly use depth as the measure, when dealing with a suitable property.

To combine the results mentioned in Theorems 22 and 23, so that we could use systems with erasing rules and nested defined functions on the right-hand sides, and still have decidable semantic matching problems, we restrict the left-hand side of the initial goal to be solved.

Definition 19 (Admissible Term). A term t is said to be *admissible* (with respect to depth) if t does not contain any defined function nested below any function which is depth decreasing.

Theorem 24. *Let R be a convergent rewrite system. If all right-hand sides for rules in R are either variables, or have a constructor at the top-level, and all right-hand sides are admissible, then the semantic matching problem is decidable for all goals of the form $t \rightarrow^? N$, where t is also admissible.*

Proof. The essential idea is to use the depth of N to bound all fruitless paths in the solution tree (tree generated by applying all possible transformation rules to a starting goal) of $t \rightarrow^? N$.

We consider application of the transformation rule Mutate on a goal of the form $t' \rightarrow^? t''[\leq M]$, where M is a measure of the right-hand side of the goal (this measure is initially computed as the depth of the ground term N, and, thereafter, every application of a transformation rule assigns a measure to the subgoals). Let $t' \equiv f(t_1, \ldots, t_n)$, where f is a defined function (if not, we could first apply some number of decomposition and imitation steps to reduce the goal to this form), and let the rule used for mutation be $f(l_1, \ldots, l_n) \rightarrow r$. The main steps are:

$$\begin{aligned}
\{t' \rightarrow^? t''[\leq M]\} \quad &\rightsquigarrow_{\mathbf{Mutate}} \quad \{t_1 \rightarrow^? l_1, \ldots, t_n \rightarrow^? l_n, r \rightarrow^? t''[\leq M]\} \\
&\rightsquigarrow_{\mathbf{Decompose}} \quad \{t_1 \rightarrow^? l_1, \ldots, t_n \rightarrow^? l_n, \\
&\qquad\qquad r_1 \rightarrow^? t_1''[\leq M-1], \ldots, r_m \rightarrow^? t_m''[\leq M-1]\}
\end{aligned}$$

Since every right-hand side, by assumption, has constructors at the top, we have shown the decomposition step which may be applied to $r \to^? t''$, assuming that the top-level constructor of r has m immediate subterms.

Let $\succ$ be a well-founded ordering on goals such that $s_1 \to^? t_1[\leq M_1] \succ s_2 \to^? t_2[\leq M_2]$ if either $M_1 > M_2$ or $M_1 = M_2$ and s_2 is a proper subterm of s_1.

There are two cases to be considered:

- If f is non-decreasing, then it must be the case that each of the t_i terms must have a normal form which is bounded by M. Therefore, we have:

$$\{t' \to^? t''[\leq M]\}$$
$$\leadsto \cdots \leadsto$$
$$\{t_1 \to^? l_1[\leq M], \ldots, t_n \to^? l_n[\leq M], r_1 \to^? t_1''[\leq M-1], \ldots, r_m \to^? t_m''[\leq M-1]\},$$

  which means that the new set of goals is smaller than the original set (containing the single goal $t' \to^? t''$), with respect to the multiset extension of $\succ$.

- If f is potentially-decreasing, then by assumption there are no defined functions in the subterms t_i, and therefore the corresponding goals can be solved using decomposition and imitation alone.

This shows that every application of the transformation rule for mutation reduces the new set of goals. It is easy to show that imitation and decomposition also decrease goals in the ordering $\succ$, and hence the result. □

Example 30. Consider the rule $eq(x, x) \to x$, together with the definition of $+$ and $*$ as in Example 20. For this system, a goal of the form $eq(x + z, z * y) \to^? 0$ is solvable.

7.2.4 Restricted Left-Linear Rules

In this section we consider possible extensions of Theorem 24 for left-linear rewrite systems, but for which the restrictions on the left-hand side of the initial goal are not required. The main problem with having nested defined functions on the right-hand sides of rules together with erasing rules is that it becomes necessary to compare non-ground terms with respect to their depth, which may not be possible in general. Here, we attempt to identify a subclass of

systems for which such comparisons can be made (essentially by forcing the variables of one term to be a subset of those in the other). Henceforth, we will assume that constants and variables have depth one. In Theorem 25 we use syntactic restrictions on the rewrite system which results in decidable matching (unlike the semantic restriction of admissibility used in Theorem 24; admissibility, in general, is an undecidable property, even for a convergent rewrite system; see Lemma 21):

Theorem 25. *Let R be a convergent left-linear rewrite system. If for every rule $f(l_1, \ldots, l_n) \rightarrow r$ in R*

1. *each $l_i, 1 \leq i \leq n$, is of depth at most two,*
2. *r is either a variable or has a constructor at the top-level, and*
3. *whenever at least one l_j has depth greater than one, r has depth greater than one,*

then the semantic matching problem is decidable for R.

Example 31. The following definition of squaring using $+$ and $*$ obeys all the syntactic restrictions of Theorem 25, and therefore has a decidable matching problem:

$$\begin{aligned}
0 + x &\rightarrow x \\
s(x) + y &\rightarrow s(x + y) \\
0 * x &\rightarrow 0 \\
x * 0 &\rightarrow 0 \\
s(x) * s(y) &\rightarrow s(y + (x * s(y))) \\
sq(0) &\rightarrow 0 \\
sq(s(x)) &\rightarrow s(sq(x) + (s(s(0)) * x))
\end{aligned}$$

We now state a proof of the theorem:

Proof. Since we have a left-linear system, we only need to solve goals of the form $s \rightarrow^? t$, where any variable x in t is linear in t and does not occur in the right-hand side of any other subgoal (this is true because we have directed goals, and terms on the right-hand sides of goals could either be left-hand sides of (left-linear) rules from previous mutations, or subterms of the

ground term N, when solving for a initial goal of the form $s' \rightarrow^? N$; see Lemma 10). Thus, we can apply a proof quite similar to that in Theorem 22, and show that the procedure is terminating for this case. (As shown in Theorem 9, it is sufficient to consider mutation and decomposition, that is, one need not consider imitation and application for completeness, for matching in theories defined by left-linear convergent systems. In the proof of Theorem 9 we used a particular selection strategy for picking subgoals to solve; we continue to use the same selection strategy for solving subgoals in this proof.)

Let $\succ$ be the well-founded ordering on goals such that $s_1 \rightarrow^? t_1 \succ s_2 \rightarrow^? t_2$ if either $depth(t_1) > depth(t_2)$ or $depth(t_1) = depth(t_2)$ and s_2 is a proper subterm of s_1.

Consider a goal of the form $f(s_1, \ldots, s_n) \rightarrow^? N[\bar{x}]$. (We use the notation $N[\bar{x}]$ to denote a term linear in variables $\bar{x}$ and for which no other subgoal in the current set has any of these variables on the right-hand side; Lemma 10 shows why considering such goals is sufficient.) We show by induction on the ordering $\succ$ that any solution to this goal is bounded in depth by that of $N[\bar{x}]$. There are several cases to be considered:

- If $N[\bar{x}]$ is a variable, then by Lemma 11, we do not have to solve this goal any further. Therefore, the only solution to this goal is an indeterminate (unbound variable) for each variable of $f(s_1, \ldots, s_n)$. Thus, the solution is of depth one, which is the same as that of $N[\bar{x}]$.

- If $N[\bar{x}]$ is a constant, then for decomposition to work, $f(s_1, \ldots, s_n)$ must be the identical constant, which gives the empty substitution as the only solution, and therefore the hypothesis holds in this case.

  Next, consider mutation of the goal $f(s_1, \ldots, s_n) \rightarrow^? N[\bar{x}]$, $N[\bar{x}]$ a constant, using a rule of the form $f(l_1, \ldots, l_n) \rightarrow r$. The only time such a rule could work is if $depth(r) = 1$. (For any other rule, by the assumption of the theorem, there has to be a constructor at the top-level of r, which would lead to failure when solving the $r \rightarrow^? N[\bar{x}]$ subgoal.) Furthermore, since r has depth one, by the assumption of the theorem, each $l_i, 1 \leq i \leq n$, must be of depth one also. If r is a constant, then the only possible solution to the goal $r \rightarrow^? N[\bar{x}]$ is the empty substitution ($\sigma = \{\}$). However, if r is a variable, say z, then this goal has a unique solution, bounded by the depth of $N[\bar{x}]$ (the solution is $\{z \mapsto N[\bar{x}]\}$). In either case, each of the subgoals $s_1 \rightarrow^? l_1\sigma, \cdots, s_n \rightarrow^? l_n\sigma$ is smaller than the original goal

$f(s_1, \ldots, s_n) \rightarrow^? N[\bar{x}]$ (since each l_i is either a variable or a constant), and the proposition follows by induction on these smaller subgoals.

- For any other case, the depth of $N[\bar{x}]$ is at least two; let $N[\bar{x}] \equiv g(N_1, \ldots, N_m)$. Were we to decompose this goal, then each of the subgoals thus generated would be smaller in the ordering $\succ$. Thus, for decomposition, the proposition holds by induction on each of the smaller subgoals. Finally, consider mutation of this goal using a rule of the form $f(l_1, \ldots, l_n) \rightarrow r$:

$$f(s_1, \ldots, s_n) \stackrel{?}{\rightarrow} N[\bar{x}] \rightsquigarrow_{\mathbf{Mutate}} s_1 \stackrel{?}{\rightarrow} l_1, \cdots, s_n \stackrel{?}{\rightarrow} l_n, r \stackrel{?}{\rightarrow} N[\bar{x}],$$

  there are further cases:

  - If we require r to be of depth one, then r must be a variable, say z. (A constant for r does not work, since the constant must be a constructor by the requirements of the theorem, and therefore, the goal $r \rightarrow^? N[\bar{x}]$ has no solution.) In this case the subgoal $r \rightarrow^? N[\bar{x}]$ (that is, $z \rightarrow^? N[\bar{x}]$) is trivially solvable, and the proposition holds for this subgoal. Furthermore, by the assumption of the theorem, each $l_i, 1 \leq i \leq n$, has depth one. Suppose σ is the (unique) solution to the goal $z \rightarrow^? N[\bar{x}]$. Therefore, as in the previous case, we have $depth(l_i\sigma) \leq depth(N[\bar{x}]), 1 \leq i \leq n$. Thus, the proposition holds by applying the hypothesis on the smaller subgoals $s_1 \rightarrow^? l_1\sigma, \cdots, s_n \rightarrow^? l_n\sigma$.
  - If r has depth greater than one, then it must have a leading constructor (by assumption of the theorem). Suppose $r \equiv g(r_1, \ldots, r_m)$, where g is a constructor. Thus, it is possible to decompose the goal $r \rightarrow^? N[\bar{x}]$ at least once, leading to smaller subgoals of the form $r_1 \rightarrow^? N_1, \ldots, r_m \rightarrow^? N_m$. Then, by applying the inductive hypothesis on these smaller subgoals, we conclude that the proposition holds for $r \rightarrow^? N[\bar{x}]$, providing a solution σ, say. Furthermore, let $depth(N[\bar{x}]) = d > 1$. If x is a variable in any $l_i, 1 \leq i \leq n$, such that x appears in r, by the inductive hypothesis, the depth of the term bound to x in σ would be at most $d - 1$ (since at least one top-level decomposition was performed on the $r \rightarrow^? N[\bar{x}]$, as indicated before). Thus,

$depth(l_i\sigma) \leq d, 1 \leq i \leq n$, and once again we can apply the inductive hypothesis on the subgoals $s_1 \rightarrow^? l_1\sigma, \cdots, s_n \rightarrow^? l_n\sigma$ to get the result.

□

Each of the restrictions used in Theorem 25 is necessary for decidability: If we drop the requirement of left-linearity, then we could get undecidability; see Section 7.2.2 for counterexamples. If we drop restriction 2 matching becomes equivalent to solving the emptiness problem for the intersection of two context-free languages, which is undecidable; see [Heilbrunner and Hölldobler, 1987] for the construction. In the remaining cases, we show that matching of certain goals would result in unification in the theories of addition ($+$) and multiplication ($*$). Notice that the definitions of $+$ and $*$ in Example 20 obey all the syntactic restrictions of Theorem 25. First of all, we relax restriction 3, that is, we allow subterms of depth greater than one below the root on the left-hand side, without requiring that the right-hand side be of depth at least two. The following example illustrates the problem:

Example 32.

$$f(1) \rightarrow 1 \tag{7.1}$$

$$f(s(1)) \rightarrow 1 \tag{7.2}$$

$$g(1,1) \rightarrow s(1) \tag{7.3}$$

$$g(s(x), s(y)) \rightarrow s(f(g(x,y))) \tag{7.4}$$

Rule 7.2 is the only one which violates the nesting criterion. It can be shown by induction that the function g has the property

$$g(x,y) = s(1) \text{ if and only if } x = y = s^n(1), n \geq 0.$$

Therefore, a goal of the form $g(s,t) \rightarrow^? s(1)$ would, in general, be undecidable, when s and t are terms involving $+$ and $*$ (see Example 20).

Finally, we relax condition 1, and allow depths greater than two below the root function on left-hand sides of rules (but in order to ensure that the last condition not be violated, we would insist that whatever depth we have on the left-hand side must show up on every path on the

right-hand side, by way of leading constructors). Consider the following example, wherein, we encode f from Example 32 using new rules:

Example 33.

$$F(s(x)) \rightarrow s(1) \tag{7.5}$$

$$G(s(s(1)), s(s(x))) \rightarrow s(s(s(x))) \tag{7.6}$$

$$g(1,1) \rightarrow s(s(1)) \tag{7.7}$$

$$g(s(x), s(y)) \rightarrow s(F(G(g(x,y), s(s(1))))) \tag{7.8}$$

Notice that none of the rules have an immediate subterm which is of greater depth than the number of leading constructors on the corresponding right-hand side. However, Rule 7.5 erases x, while Rule 7.6 is the only one which violates the depth criterion for left-hand sides (it allows immediate subterms of depth 3 on its left-hand side). It is easy to check that $F(G(s(s(1)), s(s(z)))) \rightarrow^{!} s(1)$ for a variable z (effectively, $s(s(s(z)))$ produced by rule 7.6 gets matched with the erased variable, and some of the structure gets removed in the process). Thus, similar to Example 32 we have

$$g(x,y) = s(s(1)) \text{ if and only if } x = y = s^n(1), n \geq 0.$$

For reasons mentioned in Example 32, the matching problem is undecidable for this system (together with the definitions of $+$ and $*$).

7.3 Decidable Unification

The class of matching problems that we have considered in the previous section is a simplified version of the general semantic unification problem. For instance, for the system provided in Example 31, we showed that the matching problem is decidable; however, the unification problem for this system is undecidable. Thus, it is evident that we need stronger restrictions on the rewrite systems in order to have decidable unification. We start with the following:

Theorem 26. *Let R be a convergent rewrite system, in which every right-hand side is either a variable, a ground term or a constructor-only term. Then the unification problem is decidable for R.*

Proof. We use the complete unification procedure from Section 3.1 (Table 3.2), and show that for R (as restricted above) the procedure is terminating.

Consider a goal of the form $r \rightarrow^? t$, where r is a variant (after variable renaming) of some right-hand side of a rule, and t is any term. Since r could either be a variable, a ground or constructor term, this goal can be fully solved without having to use mutation at all. (In case r is a ground term, its normal form must be unifiable with t. If r is a variable, we could either eliminate or bind. Finally, if r is a constructor-only term, it must be syntactically unifiable with t, which, in our case, is checked using some number decomposition and imitation steps, followed by the use of Eliminate and Bind.) Thus, all solutions to this goal can be generated in finite time.

Whenever we have a goal of the form $s \rightarrow^? t$, for any term s, we could use one of Eliminate, Bind, Mutate, Decompose or Imitate from Table 3.2. Let $\succ$ be the well-founded ordering on goals such that $s \rightarrow^? t \succ s' \rightarrow^? t'$ if s' is a proper subterm of s. We use the (well-founded) multiset extension of this ordering, which we also denote as $\succ$. It is evident that any application of Eliminate, Bind, Decompose and Imitate cause a reduction in the ordering. For mutation we have:

$$s \rightarrow^? t \quad \leadsto_{\mathbf{Mutate}} \quad s_1 \rightarrow^? l_1, \cdots, s_n \rightarrow^? l_n, r \rightarrow^? t.$$

By previous argument, all solutions to the goal $r \rightarrow^? t$ can be generated in finite time. Thus, we could replace the derivation sequence above with finitely many (since the number of solutions to $r \rightarrow^? t$ is finite) sequences of the form:

$$\begin{array}{lll} s \rightarrow^? t & \leadsto_{\mathbf{Mutate}} & s_1 \rightarrow^? l_1, \cdots, s_n \rightarrow^? l_n, r \rightarrow^? t \\ & \leadsto^* & s_1 \rightarrow^? l_1, \cdots, s_n \rightarrow^? l_n, \sigma \end{array}$$

(we have one such sequence for every solution (σ) to the goal $r \rightarrow^? t$). Thus, for every such sequence emanating from a mutation, we now have a decrease in $\succ$. □

Theorem 26 is a slight extension of one in [Hullot, 1980], which considers only ground terms or variables on the right-hand sides.

For the next result we need the following definition:

Definition 20 (Subterm Composing). A rewrite rule $l \rightarrow r$ is *subterm composing* if every subterm of r with a defined function at its root is a proper subterm of l.

Theorem 27. *The semantic unifiability problem is decidable for a convergent rewrite system in which every rule is subterm composing.*

Proof. Consider a goal of the form $r \rightarrow^? t$, as before. We show that all solutions to this goal can be generated in finite time.

By the requirements of the theorem, r is of the form $C[r_1, \ldots, r_n]_P$, where P is a set of n disjoint positions in r, the subterms rooted at which have a defined function at the top-position (the subterms are $r_1, \ldots, r_n$), and C is the constructor-only context. By assumption, each r_i is a subterm of l; hence, whenever the rule $l \rightarrow r$ is used for mutation, we would have irreducibility predicates corresponding to each r_i (or a term which contains r_i as a proper subterm). Thus, further mutation of a subgoal which has any r_i as its left-hand side would only lead to reducible (and thus redundant) solutions, which would imply that all such subgoals can be solved (for irreducible solutions) without applying mutations. Furthermore, since C is a constructor-only context, the entire goal $r \rightarrow^? t$ can be solved without further mutation, and thus has only a finite number of solutions.

The rest of the proof is similar to the one for Theorem 26. □

Theorem 27 is an extension of a decidability result in [Kapur and Narendran, 1987], which deals with rewrite systems in which every right-hand side is a subterm of the corresponding left-hand side (that is, for their rules, the constructor context C in Theorem 27 is always empty, P is the singleton set containing the root position, and $r_1 \equiv r$ is a subterm of l). However, they also show that the problem is NP-complete in their case.

In general, whenever a convergent system allows nested functions (either defined functions or constructors; with the exception that leading constructors never cause a problem) on the right-hand side, it is possible to have undecidable unification. We, therefore, will only allow the following kind of terms in right-hand sides of rules:

Definition 21 (Flat Term). A term is said to be *flat* if it has a single defined function, and all variables below this function symbol appear directly below it.

For example, if f is a defined function and s, a and 0 are constructors, then the terms $s(f(x,y))$ and $s(f(a(0,0),x))$ are flat, while $f(s(x),y)$ (x does not appear directly below f) and $a(f(x,0),f(0,y))$ (there are more than one defined function) are not.

Theorem 28. *Let R be a (left- and right-) linear convergent rewrite system. If for every function f defined by R there is at most one rule with a right-hand side that is neither a constructor term nor a variable and that is flat, then the semantic unification problem is decidable for R.*

We need the following lemma:

Lemma 29. *For R as defined in Theorem 28, all solutions to a goal $r \rightarrow^? t$ (where r is a variant of a right-hand side) can be expressed as a recurrent scheme.*

We will require the following definitions for the proof of Lemma 29:

Definition 22 (Simple Substitution). Let $X = \{x_1, \ldots, x_n\}$ and $X' = \{x'_1, \ldots, x'_n\}$ be two sets of (pairwise) distinct variables. A substitution θ is said to be a *simple substitution* if:

- The domain of θ is identical to X.
- $\theta(x_i) = t_i[X'_i], 1 \leq i \leq n$, where t_i is a context, and X'_i is a subset of X'.

Definition 23 (Iterated-Substitution). Let θ be a simple substitution from X to X' as above, and let $X'' = \{x''_1, \ldots, x''_n\}$ be set of new and distinct variables. Let θ' be the substitution which is constructed from θ by uniformly renaming (in parallel) each variable in X by the corresponding one in X', and each variable in X' by the corresponding one in X'', and by renaming all other variables using new ones. Then we define $\theta^2 = \theta \circ \theta'$. This notion can be extended to define $\theta^3, \cdots, \theta^n$ for any integer n.

For example, $\theta = \{x_1 \mapsto c(x'_1, x'_2, y), x_2 \mapsto x'_2\}$ is a simple substitution, and $\theta^2 = \{x_1 \mapsto c(c(x''_1, x''_2, y'), x''_2, y), x_2 \mapsto x''_2\}$ is an iterated substitution (θ^2 is θ iterated to its second power).

Proof. (**Of Lemma** 29) Consider the goal $r \rightarrow^? t$. The interesting case is when we have to apply the transformation rule for mutation. (Without mutation and with r flat, we get a finite branch of the solution tree, which is not a problem.) Furthermore, mutation using rules for which the right-hand sides are either constructor terms or variables leads to finite branches

(see Theorem 26 above). If r has any leading constructors, then we can use Decompose and Imitate as required, and end up with a goal of the form $f(r_1, \ldots, r_n) \rightarrow^? t'$, where f is a defined function. Consider mutation of this goal using the rule $f(l_1, \ldots, l_n) \rightarrow r'$. By the assumptions of the theorem, each r_i is either a variable or a ground term. Therefore, all the subgoals of the form $r_i \rightarrow^? l_i$ can be solved without mutation, which leaves a single subgoal of the form $r' \rightarrow^? t'$. Furthermore, because of flatness and linearity, no variable in r' could have been bound when solving the $r_i \rightarrow^? l_i$ subgoals. Also, along any path in the solution tree, the right-hand sides of goals cannot get any more complicated (mutation keeps the right-hand side intact; decomposition reduces the structure and imitation also keeps the right-hand side structure intact, since it replaces a variable with a new one for each subgoal).

Therefore, if there is an infinite path in the solution tree it must contain a repetition of the form:

$$g_1 \equiv \{r_1 \rightarrow^? t_1\} \quad \rightsquigarrow_{\mathbf{Mutate}^*} \quad \{r_2 \rightarrow^? t_2\} \equiv g_2,$$

where r_2 and t_2 are renamed versions of r_1 and t_1, respectively. In this situation we can express finitely a complete set of solutions to g_1 without having to explore g_2 further. Let σ denote the substitution generated in the path from g_1 to g_2, and σ' be any solution along some other finite path to g_1. (Since there is a single flat rule, there can be no other infinite paths.) Then, corresponding to σ', we get a complete set of solutions, which can be expressed as $\sigma^n \circ \sigma'$, $n \geq 0$ (σ^n stands for σ iterated n times, where n is an integer variable; refer to the example below for details). □

We are now ready to state a proof of Theorem 28:

Proof. In Lemma 29 we have shown that all solutions to a goal of the form $r \rightarrow^? t$ can be finitely generated, where r is a variant of a right-hand side of a rule in R.

To prove decidability of unification, we consider a goal of the form $s \rightarrow^? t$, for any two terms s and t. As before, any application of Eliminate, Bind, Decompose or Imitate would result in simpler subgoals. For mutation, we use the following strategy: we keep the $r \rightarrow^? t$ goal intact, and solve any of the other goals. Therefore, along any path in the solution tree, we would either have no remaining goals (if Mutation was never used along that path), or a collection of subgoals, each of the form $r \rightarrow^? t$, where r is a variant of the right-hand side of a rule in R.

By Lemma 29, we can express all solutions to any goal of the form $r \rightarrow^? t$ as a recurrent schemata. Therefore, whenever we have multiple such subgoals to solve, we have to find the language in the intersection of two such recurrent schemata, which can be done, using a result from [Socher-Ambrosius, 1993], which itself is an extension of a theorem from [Comon, 1992] (see discussion after Example 35). □

Example 34. Consider the rewrite system for addition:

$$0 + x \rightarrow x \tag{7.9}$$

$$s(x) + y \rightarrow s(x + y) \tag{7.10}$$

The goal

$$y + (y + z) \stackrel{?}{=} z + z$$

is transformed into two directed goals, $\{y + (y + z) \rightarrow^? x', z + z \rightarrow^? x'\}$. Some of the derivation steps are:

$$\left\{ \begin{array}{lll} y + (y + z) & \rightarrow^? & x' \\ z + z & \rightarrow^? & x' \end{array} \right\} \rightsquigarrow_{\textbf{Mutate(7.10)}} \left\{ \begin{array}{c} y \rightarrow^? s(x_1), y + z \rightarrow^? y_1, \\ s(x_1 + y_1) \rightarrow^? x', z + z \rightarrow^? x' \end{array} \right\}$$

We can solve the goal $z + z \rightarrow^? x'$ by changing it to $z_1 + z_2 \rightarrow^? x', z_1 = z_2$. (This is how we can handle non-linear variables in the original goal in general.) The steps involved in solving the goal $z_1 + z_2 \rightarrow^? x'$ are:

$$\begin{array}{lll} \{z_1 + z_2 \rightarrow^? x'\} & \rightsquigarrow_{\textbf{Mutate(7.9)}} & \{z_1 \rightarrow^? 0, z_2 \rightarrow^? z_0, z_0 \rightarrow^? x'\} \\ & \rightsquigarrow & \{z_1 \mapsto 0, z_2 \mapsto x'\} \\ \{z_1 + z_2 \rightarrow^? x'\} & \rightsquigarrow_{\textbf{Mutate(7.10)}} & \{z_1 \rightarrow^? s(x_2), z_2 \rightarrow^? y_2, s(x_2 + y_2) \rightarrow^? x'\} \\ & \rightsquigarrow & \{x_2 + y_2 \rightarrow^? x'', \theta\} \end{array}$$

where $\theta \equiv \{z_1 \mapsto s(x_2), z_2 \mapsto y_2, x' \mapsto s(x'')\}$.

We now have a subsuming pattern of the form

$$\{z_1 + z_2 \rightarrow^? x'\} \quad \rightsquigarrow^* \quad \{x_2 + y_2 \rightarrow^? x''\}$$

producing the solution $z_1 = s^n(0), z_2 = z_0, x' = s^n(z_0)$, which, together with the constraint $z_1 = z_2$, simplifies to $z = s^n(0), x' = s^{2n}(0)$.

Similarly, we can simplify the remaining goals to get the following: $x_1 = s^m(0), y_1 = y_0, x' = s^{m+1}(y_0)$ (as a general solution to the goal $s(x_1 + y_1) \rightarrow^? x'$), $y = s^j(0), z = z_a, y_1 = s^j(z_a)$ (from the goal $y + z \rightarrow^? y_1$), and $y = s(x_1)$.

We solve these equations (with respect to n, m and j) to get $y = s^j(0), z = s^n(0)$, where $x' = s^{2j+n}(0) = s^{2n}(0)$, which gives $2j = n$ after simplification.

Example 35. The following rewrite system (*append* and *interleave* on lists) has a decidable semantic unification problem:

$$\begin{aligned}
append(nil, x) &\rightarrow x \\
append(x \cdot y, z) &\rightarrow x \cdot append(y, z) \\
\\
interleave(nil, x) &\rightarrow x \\
interleave(x, nil) &\rightarrow x \\
interleave(x \cdot y, z) &\rightarrow x \cdot interleave(z, y)
\end{aligned}$$

In Example 34, we handled non-linear variables in the original goal by solving additional constraints, which was simple because we had a unary function (s) to deal with. We now briefly indicate how solutions can be represented and such constraints solved in general. According to the requirements of Theorem 28, if x is a variable in the original goal, then along the infinite branch of the solution tree, we have the following:

$$x \mapsto u[x_1], x_1 \mapsto v[x_2], x_2 \mapsto v[x_3], \ldots,$$

where v is the context which gets repeated due to the subsuming pattern. Thus, in effect, the variable x gets bound to a term-pattern, which we represent as $x \mapsto u[v^n[x_{n+1}]]$. Therefore, in solving a constraint of the form $x = y$, where both x and y could be bound to such term patterns, we have to "unify" these patterns. These patterns are exactly the ones considered in [Socher-Ambrosius, 1993, Section 4]. In fact, our patterns are slightly simpler than the ones handled by [Socher-Ambrosius, 1993], because we have terms which could be iterated along a single path alone, and additionally, all the variables—*flexible-variables* in their terminology—

are linear in our case. The claim in Section 4 of [Socher-Ambrosius, 1993] is that the unification algorithm is terminating in the general case (that is, even with non-linear flexible variables) using the same proof which is sketched in Section 3 (for a similar system without flexible variables) of [Socher-Ambrosius, 1993]. However, this does not seem to be correct, since the termination proof makes use of the fact that the number of variables does not increase when Merge (similar to Bind in our case) is applied, which is not true, since Merge may introduce new flexible variables in the rest of the equations. However, if the flexible-variables are assumed to be linear (as we require), then there is no problem, since we would never have to merge flexible variables.

The requirement that every rule should be left- and right-linear for Theorem 28 seems to be overly restrictive. It may be possible to use a different language (or even an extension of the one in [Socher-Ambrosius, 1993]) to relax these restrictions. However, for decidability, the restriction that each function symbol may have at most one rule with a flat right-hand side is necessary:

Lemma 30. *There is no decision procedure for the matching problem in a (left- and right-) linear convergent rewrite system, in which every right-hand side is either a variable or a constructor term, or is flat.*

Proof. We show that semantic matching in such theories can be used to simulate the undecidable Post's Correspondence Problem (PCP, [Hopcroft and Ullman, 1979]).

An instance of PCP consists of two lists $A = w_1, \ldots, w_k$ and $B = x_1, \ldots, x_k$, of strings over some alphabet Σ. This instance has a *solution* if there exists a sequence of integers $i_1, \ldots, i_m, m \geq 1$, such that

$$w_{i_1}, \ldots, w_{i_m} = x_{i_1}, \ldots, x_{i_m}.$$

The following example illustrates how semantic matching can be used to generate solutions to a particular instance of the Post's Correspondence Problem:

Example 36. Let $\Sigma = \{s, p\}$, while A and B are as given below:

	List A	List B
i	w_i	x_i
1	s	sss
2	$spsss$	sp
3	sp	p

We construct the following rewrite system R:

$$\begin{aligned}
eq(s(x), s(y)) &\rightarrow eq(x, y) \\
eq(p(x), p(y)) &\rightarrow eq(x, y) \\
firsts(nil) &\rightarrow nil \\
firsts(1 \cdot x) &\rightarrow s(firsts(x)) \\
firsts(2 \cdot x) &\rightarrow s(p(s(s(s(firsts(x)))))) \\
firsts(3 \cdot x) &\rightarrow s(p(firsts(x))) \\
snds(nil) &\rightarrow nil \\
snds(1 \cdot y) &\rightarrow s(s(s(snds(y)))) \\
snds(2 \cdot y) &\rightarrow s(p(snds(y))) \\
snds(3 \cdot y) &\rightarrow p(snds(y))
\end{aligned}$$

It is easy to see that this instance of PCP has a solution if and only if the goal

$$eq(firsts(x \cdot y), snds(x \cdot y)) \stackrel{?}{=} eq(nil, nil)$$

is satisfiable.

From the construction, it is evident that, given any instance of PCP, we can similarly construct a convergent rewrite system with the required syntactic restrictions, such that the matching problem described above has a solution if and only if the instance of PCP under consideration has one. □

7.4 Summary on Decidability Results

Most of the results on decidable semantic matching using non-erasing rules were first presented in [Dershowitz *et al.*, 1992], while the theorem for decidable unification (using left- and right-linear systems with flat right-hand sides) was presented in [Dershowitz and Mitra, 1993].

Decidable matching and unification are particularly useful in pattern directed languages, constraint solving and theorem proving. In this chapter we have studied these problems for convergent systems and have shown the there exists different complex characterization (based on syntactic as well as semantic properties) which provide decidable subproblems. The results that we provide for matching are "tight," in the sense that whenever any of the conditions that we impose is violated, the resulting rewrite system may have undecidable matching (unification) problems. Before our work, almost no such characterization was known for decidable matching problems, and the ones for unification were very restrictive.

At the end of Section 7.3 we mentioned a problem about a result from [Socher-Ambrosius, 1993]. We believe that a correct proof of termination of the unification procedure mentioned therein would help generalize decidability results for systems with flat right-hand sides (by allowing non-linear variables in rules). Another approach for a recurrent schematization of an infinite family of terms (using the so called *primal grammars*) has been mentioned in [Hermann, 1992], wherein a narrowing-like strategy has been used for unification of such families of terms. However, a proof of termination of this procedure (in the most general case of primal systems, including *marked variables*) is still forthcoming.

8 SUMMARY AND FUTURE WORK

Semantic unification is of importance in programming language interpreters and theorem provers. In this thesis we have studied the problem of unification in theories defined by convergent rewrite systems. After proving completeness for the resulting unification procedure, we have discussed several interesting extensions of the problem, and have shown that a complete set of semantic matchings can be found using a simpler set of transformations for convergent systems that admit either a left-linear or a non-erasing presentation. Furthermore, we have used the different transformation systems to derive syntactic and semantic restrictions on the rewrite system that result in the unification and matching problems being decidable. Solvable matching is useful in pattern-directed languages, while solvable unification can be used in inductive theorem proving. We now briefly mention some possible extensions of the results presented here.

8.1 Unification in Combined Theories

Consider the very general method for proving equality of terms (that is, a method for solving the validity problem), using the inference system of Table 8.1. If we were to use the transformation

Reflect	$\dfrac{}{s \rightarrow^* s}$
Axiom	$\dfrac{l \rightarrow r \in R, r\sigma \rightarrow^* t}{l\sigma \rightarrow^* t}$
Decompose	$\dfrac{s_1 \rightarrow^* t_1, \ldots, s_n \rightarrow^* t_n, f(t_1, \ldots, t_n) \rightarrow^* u}{f(s_1, \ldots, s_n) \rightarrow^* u}$

Table 8.1: Inference rules for validity of $\rightarrow^*$

system of Table 8.1, we could use rewrite systems (R in Axiom of Table 8.1) that are non-terminating or non-confluent (or both).

A very natural question in this general framework is that of finding a basis set of unifiers. Notice that the unification problem in this setup is a generalization of E-unification in the sense

Imitate(l)	$x \rightarrow^? g(t_1, \ldots, t_n)$ $\leadsto$ $x = g(x_1, \ldots, x_n), x_1 \rightarrow^? t_1, \ldots, x_n \rightarrow^? t_n$
Imitate'	$x \rightarrow^? y$ $\leadsto$ $x = g(x_1, \ldots, x_n), y = g(y_1, \ldots, y_n), x_1 \rightarrow^? y_1, \ldots, x_n \rightarrow^? y_n$
Var-Param	$x \rightarrow^? y$ $\leadsto$ $x = f(x_1, \ldots, x_n), x_1 \rightarrow^? l_1, \ldots, x_n \rightarrow^? l_n, r \rightarrow^? y$ where $f(l_1, \ldots, l_n) \rightarrow r$ is a renamed rule in R

Table 8.2: Transformation rules for satisfiability of $\rightarrow^?$

of [Gallier and Snyder, 1989; Snyder, 1991], since, in our system, we would get E-unification if we used both variants ($l \rightarrow r$ and $r \rightarrow l$) of every equation $l = r \in E$. It can be shown that when the transformation rules of Table 8.2 are used, together with those from Table 3.2, we get a complete unification procedure in this general setting. One problem with this system is that the transformation rules (essentially the rules from Table 8.2) are highly non-deterministic. In fact, even with a simple goal of the form $x \rightarrow^? y$ it is possible to have an infinite subsuming derivation, a problem quite similar to the one discussed by Snyder [1991, page 65], for a system for general E-unification. However, if we were to consider irreducible solutions alone, it can be shown that the rules from Table 3.2 are sufficient for the purpose (that is, Imitate(l), Imitate' and Var-Param are redundant in this case). Therefore, an interesting line of research would be to see if a more efficient system can be developed for this problem. Another interesting problem is to come up with examples which illustrate that each of the transformation rules in Table 8.2 is indeed necessary for completeness, without the assumption of irreducible solutions.

8.2 Higher-Order Matching and Unification

In Chapter 4 we looked at a way of combining higher-order features with a first-order convergent rewrite system to get a complete set of transformations for higher-order unification. Two related problems ought to be addressed:

- We have only considered typed combinatory logic, which gives us a system $\mathcal{C}$ (see Section 4.1) that is convergent. In general (considering the untyped calculus) $\mathcal{C}$ would only be confluent (but non-terminating). It would be interesting to see if the combination technique of Chapter 4 would still be valid in this case.

- The second problem concerns decidability of higher-order matching. Higher-order matching seems to be significantly simpler than higher-order unification [Baader and Siekmann, 1993]. Huet [1975] himself showed that second-order matching is decidable and conjectured that this decidability result holds in general, which is still an open problem, except for the third-order case [Dowek, 1992]. It may be easier to formulate the higher-order matching problem in combinatory logic and get a better notion of termination of the resulting system of transformation rules, using the completeness result from Section 3.5 together with techniques from Chapter 7.

Furthermore, our solution for unification in combinatory-logic is, in some sense, very similar to the solution for the unification problem in general associative-commutative theories. In either case, we have a convergent system, together with some additional axioms which are handled outside the basic unification procedure. For the former case, we have to apply the extensionality axiom non-deterministically, while in the latter case, we use the axioms of associativity and commutativity (by way of rearranging the subterms). It may be possible to generalize these two results, so that we could solve the unification problem modulo any general equational theory.

8.3 Associative-Commutative Reduction Orderings

In Chapter 6 we have discussed a simple restriction on the recursive path ordering which results in a relation that can be used to prove termination in the presence of associative-commutative function symbols. Recently, Rubio and Nieuwenhuis [1993] have extended a similar ordering to one that is total on (non-AC equivalent) ground terms. However, their ordering orients the distributivity axiom the "wrong" way. It would be interesting to see if a combination of our ordering and the one proposed in [Rubio and Nieuwenhuis, 1993] would result in an AC ordering which has the advantages of both.

Another important problem, which has not been addressed in this thesis, is that of proving ground convergence. Our completeness results require the rewrite system to satisfy this (somewhat) weaker notion of convergence, that is, we only require that the system be convergent on ground terms, and not on all terms. In fact, in the programming context, function definitions usually satisfy this requirement. Unfortunately, only overly restrictive or ad-hoc schemes are known for proving ground convergence of a set of axioms, and the problem is known to be undecidable even for terminating rewrite systems. Therefore, another important line of research would be to find practically useful sufficient conditions for this property.

A EXAMPLES USING GOAL-DIRECTED APPROACH

In this appendix, illustrative examples using the top-down equation solving procedure are included. The transcripts are taken using the software, SUTRA, which is a Common Lisp version of RRL (Rewrite Rule Laboratory) [Kapur and Zhang, 1987]. The parts of this system which interest us are:

1. Completion procedure for converting an equational system into a set of rewrite rules,
2. Special completion for theories with AC-functions,
3. Rewriting techniques for deriving the normal form of terms modulo a conditional theory,
4. Top-down goal directed equation solving procedure, and
5. Goal-directed equation solver for AC theories as discussed in Section 5.1, with the restriction that all AC functions are completely defined.

A.1 Factorial of Natural Numbers

In this section we use conditional rules for defining *factorial*, and show how goals can be solved in the resulting convergent system.

```
/*************************************************************/
/*                                                           */
/*    EXAMPLE  I                                             */
/*      Factorial of Natural Numbers                         */
/*                                                           */
/*************************************************************/

RRL-> auto fact

Equations read in are:
```

```
 1. (0 < 0) == FALSE  [USER, 1]
 2. (0 < S(0))  [USER, 2]
 3. (S(X) < S(Y)) == (X < Y)  [USER, 3]
 4. (0 < S(X)) == TRUE  if  (0 < X)  [USER, 4]
 5. (S(X) - S(Y)) == (X - Y)  [USER, 5]
 6. (X - 0) == X  [USER, 6]
 7. (0 + X) == X  [USER, 7]
 8. (S(X) + Y) == S((X + Y))  [USER, 8]
 9. (0 * X) == 0  [USER, 9]
10. (S(X) * Y) == (Y + (X * Y))  [USER, 10]
11. FACT(0) == S(0)  [USER, 11]
12. FACT(X) == (X * FACT((X - S(0))))  if  (0 < X)  [USER, 12]

Using Conditional Rewriting Method ...

Your system is possibly canonical.

  [1] (0 < 0) ---> FALSE  [USER, 1]
  [2] (0 < S(0)) ---> TRUE  [USER, 2]
  [3] (S(X) < S(Y)) ---> (X < Y)  [USER, 3]
  [4] (0 < S(X)) ---> TRUE  if   { (0 < X) }   [USER, 4]
  [5] (S(X) - S(Y)) ---> (X - Y)  [USER, 5]
  [6] (X - 0) ---> X  [USER, 6]
  [7] (0 + X) ---> X  [USER, 7]
  [8] (S(X) + Y) ---> S((X + Y))  [USER, 8]
  [9] (0 * X) ---> 0  [USER, 9]
 [10] (S(X) * Y) ---> (Y + (X * Y))  [USER, 10]
 [11] FACT(0) ---> S(0)  [USER, 11]
 [12] FACT(X) ---> (X * FACT((X - S(0))))  if   { (0 < X) }   [USER, 12]

Time used                          =   1.08 sec
```

```
Number of rules generated           = 12
Number of rules retained            = 12
Number of critical pairs            = 1
Time spent in normalization         =   0.03 sec  (3.0 percent of time)
Time spent in unification           =   0.03 sec   (3.0 percent of time)
Time spent in ordering              =   0.08 sec   (7.0 percent of time)
Time spent in simplifying the rules  =   0.40 sec   (36.0 percent of time)
Total time (including 'undo' action) = 67.0 sec

RRL-> sol fact(x) == s(0)

 Solving FACT(X) == S(0)  [USER, 13]
   X  |--> (0)
Found at Depth 2.
Time taken for this solution :   0.02 secs
Try Again?(Y,N) y
   X  |--> (S (0))
Found at Depth 7.
Time taken for this solution :   1.87 secs
Try Again?(Y,N) n
 Total Time used for solving =     1.88 seconds
RRL-> sol fact(x + y) == s(s(0))
 Solving FACT((X + Y)) == S(S(0))  [USER, 14]
   X  |--> (0)
   Y  |--> (S (S (0)))
Found at Depth 14.
Time taken for this solution :   3.60 secs
Try Again?(Y,N) y
   Time taken for this depth is :   29.18 secs
   Incrementing depth bound to 19
   X  |--> (S (0))
```

```
   Y  |--> (S (0))
Found at Depth 17.
Time taken for this solution :  63.72 secs
Try Again?(Y,N) y
   X  |--> (S (S (0)))
   Y  |--> (0)
Found at Depth 18.
Time taken for this solution :   7.68 secs
Try Again?(Y,N) n

 Total Time used for solving =    76.47 secs
```

A.2 Addition and Multiplication of Natural Numbers

In this section we use the system for addition and multiplication to solve a goal of the form $x * y \stackrel{?}{=} s(s(s(s(0))))$ (which is very similar to the goal we considered in Example 11). As mentioned therein, for this system, it is possible to finitely generate all solutions to matching goals as above, provided we use inductive consequences (Rules [2] and [5] below) for simplification. The program allows the user to provide a list of rules which will not be used for mutation (see Section 3.3.2 for correctness issues); in this case, we neglect the same two rules.

```
/*************************************************************/
/*                                                           */
/*    EXAMPLE  II                                            */
/*      Addition and Multiplication of Natural Numbers       */
/*                                                           */
/*************************************************************/

RRL-> auto sol.cmd
```

```
Equations read in are:
 1. (0 + X) == X  [USER, 1]
 2. (X + 0) == X  [USER, 2]
 3. (S(X) + Y) == S((X + Y))  [USER, 3]
 4. (0 * X) == 0  [USER, 4]
 5. (X * 0) == 0  [USER, 5]
 6. (S(X) * Y) == (Y + (X * Y))  [USER, 6]

Your system is canonical.

  [1] (0 + X) ---> X  [USER, 1]
  [2] (X + 0) ---> X  [USER, 2]
  [3] (S(X) + Y) ---> S((X + Y))  [USER, 3]
  [4] (0 * X) ---> 0  [USER, 4]
  [5] (X * 0) ---> 0  [USER, 5]
  [6] (S(X) * Y) ---> (Y + (X * Y))  [USER, 6]

Time used                           =   0.08 sec
Number of rules generated           = 6
Number of rules retained            = 6
Number of critical pairs            = 3
Time spent in normalization         =   0.00 sec  (0.0 percent of time)
Time spent in unification           =   0.00 sec   (0.0 percent of time)
Time spent in ordering              =   0.02 sec   (20.0 percent of time)
Time spent in simplifying the rules  =   0.02 sec   (20.0 percent of time)
Total time (including 'undo' action) = 30.0 sec

RRL-> br
Break: to LISP. Type (rrl) to resume.
Broken at START-UP.  Type :H for Help.
RRL>>( show-res-rules $rule-set)
```

```
Rule 1 is -         (+ (0) X) --> X
Rule 2 is -         (+ X (0)) --> X
Rule 3 is -         (+ (S X) Y) --> (S (+ X Y))
Rule 4 is -         (* (0) X) --> (0)
Rule 5 is -         (* X (0)) --> (0)
Rule 6 is -         (* (S X) Y) --> (+ Y (* X Y))
Give rule numbers not needed for restructuring
           Enter NIL if all rules are required    (2 5)
 (2 5)
RRL>>(rrl)
RRL-> sol x * y == s(s(s(s(0))))
 Solving (X * Y) == S(S(S(S(0))))  [USER, 7]
   X  |--> (S (S (0)))
   Y  |--> (S (S (0)))
Found at Depth 9.
Time taken for this solution :  35.48 secs
Try Again?(Y,N) y
   X  |--> (S (S (0)))
   Y  |--> (S (S (0)))
Found at Depth 11.
Time taken for this solution :   1.18 secs
Try Again?(Y,N) y
   X  |--> (S (0))
   Y  |--> (S (S (S (S (0)))))
Found at Depth 8.
Time taken for this solution :   0.65 secs
Try Again?(Y,N) y
   X  |--> (S (0))
   Y  |--> (S (S (S (S (0)))))
Found at Depth 9.
Time taken for this solution :   2.53 secs
```

```
Try Again?(Y,N) y
   X  |--> (S (0))
   Y  |--> (S (S (S (S (0)))))
Found at Depth 10.
Time taken for this solution :   1.78 secs
Try Again?(Y,N) y
   X  |--> (S (S (0)))
   Y  |--> (S (S (0)))
Found at Depth 10.
Time taken for this solution :   3.22 secs
Try Again?(Y,N) y
   X  |--> (S (S (0)))
   Y  |--> (S (S (0)))
Found at Depth 12.
Time taken for this solution :   1.35 secs
Try Again?(Y,N) y
   X  |--> (S (S (S (S (0)))))
   Y  |--> (S (0))
Found at Depth 14.
Time taken for this solution :   3.22 secs
Try Again?(Y,N) y

No more solutions to this equation.

 Total Time used for solving =    49.42 seconds
```

A.3 Addition and Multiplication of Natural Numbers (AC)

This time around, we use the AC axioms for addition and multiplication. The implementation, in this case, adheres to the transformation rules described in Section 5 of [Dershowitz *et al.*,

1990], rather than the ones in Table 5.1. Notice that the latter system is more efficient, since it looks at AC-goals just like the non-AC ones, unlike the former, where we used flattening, and therefore had more positions to apply rules.

In the body of the example, we, at one point, ignore the distributivity rule (for mutation). This rule is required for AC-completion to work, however, since the left-hand side of Rule [5] $x * (y + z)$ has a defined function in a proper subterm (and would therefore produce only reducible solutions through mutation); see discussion in Section 5.1.

```
/*************************************************************/
/*                                                           */
/*    EXAMPLE  III                                           */
/*      Addition and Multiplication of Natural Numbers (AC)  */
/*                                                           */
/*************************************************************/
RRL-> auto test
Equations read in are:
 1. (0 + X) == X  [USER, 1]
 2. (X + 0) == X  [USER, 2]
 3. (S(X) + Y) == S((X + Y))  [USER, 3]
 4. (0 * X) == 0  [USER, 4]
 5. (X * 0) == 0  [USER, 5]
 6. (S(X) * Y) == ((X * Y) + Y)  [USER, 6]
 7. (X * (Y + Z)) == ((X * Y) + (X * Z))  [USER, 7]
ACOPERATOR
* +
'*' is associative & commutative now.
'+' is associative & commutative now.
Your system is canonical.
  [1] (X + 0) ---> X  [USER, 1]
  [2] (Y + S(X)) ---> S((X + Y))  [USER, 3]
  [3] (X * 0) ---> 0  [USER, 4]
  [4] (Y * S(X)) ---> (Y + (X * Y))  [USER, 6]
```

```
  [5] (X * (Y + Z)) ---> ((X * Y) + (X * Z))  [USER, 7]

Time used                            =   6.63 sec
Number of rules generated            = 5
Number of rules retained             = 5
Number of critical pairs             = 90
Number of unblocked critical pairs   = 43
Time spent in normalization          =   4.72 sec  (71.0 percent of time)
Time spent in unification            =   0.95 sec   (14.0 percent of time)
Time spent in ordering               =   0.02 sec   (0.0 percent of time)
Time spent in simplifying the rules  =   0.03 sec   (0.0 percent of time)
Time spent in blocking               =   0.17 sec   (2.0 percent of time)
Total time (including 'undo' action) = 400.0 sec

RRL-> sol x + y + z == s(0)
 Solving (X + Y + Z) == S(0)  [USER, 9]
   X  |--> (S (0))
   Y  |--> (0)
   Z  |--> (0)
Found at Depth 4.
Time taken for this solution :   0.13 secs
Try Again?(Y,N) y
   X  |--> (0)
   Y  |--> (S (0))
   Z  |--> (0)
Found at Depth 5.
Time taken for this solution :   0.37 secs
Try Again?(Y,N) y
   X  |--> (0)
   Y  |--> (0)
   Z  |--> (S (0))
```

```
Found at Depth 5.
Time taken for this solution :   0.07 secs
Try Again?(Y,N) y

... ... (some stuff deleted) ... ...

No more solutions to this equation.
 Total Time used for solving =     1.38 seconds

RRL-> br
Break: to LISP. Type (rrl) to resume.
Broken at RRL.
RRL>>>(show-res-rules $rule-set)
Rule 1 is -         (+ X (0)) --> X
Rule 2 is -         (+ Y (S X)) --> (S (+ X Y))
Rule 3 is -         (* X (0)) --> (0)
Rule 4 is -         (* Y (S X)) --> (+ Y (* X Y))
Rule 5 is -         (* X (+ Y Z)) --> (+ (* X Y) (* X Z))
Give rule numbers not needed for restructuring
           Enter NIL if all rules are required    (5)
(5)
RRL>>>(rrl)
RRL-> sol x * y == s(0)
 Solving (X * Y) == S(0)  [USER, 10]
   X  |--> (S (0))
   Y  |--> (S (0))
Found at Depth 4.
Time taken for this solution :   0.07 secs
Try Again?(Y,N) y
   X  |--> (S (0))
   Y  |--> (S (0))
```

```
Found at Depth 5.
Time taken for this solution :   0.12 secs
Try Again?(Y,N) y

... ... (some stuff deleted) ... ...

No more solutions to this equation.
 Total Time used for solving =     2.17 seconds
```

A.4 Sorting Lists of Natural Numbers

In this section we show an example of sorting lists of numbers. We use a system for sorting lists of numbers into an increasing order. Although it is possible to define sorting using conditional rules, in this example we only use unconditional ones (therefore, we need the auxiliary functions *max* and *min*).

We use unification to generate all solutions to the goal $sort(x) \stackrel{?}{=} 0 \cdot s(0) \cdot s(s(0)) \cdot nil$. As discussed in Section 7.2.2, the matching problem is decidable in the theory under consideration, and, in fact, the program stops after enumerating all possible solutions to the goal.

In the example below, at one point, we have introduced a marker to show that the program may repeat solutions. This is so since an iterative depth-first search has been used to generate the solution space. By default, the program starts with some predefined value of depth, and searches the tree until such point. However, if after the specified depth, certain paths still have to be explored, the depth bound is incremented, and the search starts again at the root, thus generating duplicate solutions.

```
/*************************************************************/
/*                                                           */
/*    EXAMPLE  IV                                            */
/*      Sorting Lists of Natural Numbers                     */
/*                                                           */
/*************************************************************/
```

```
RRL-> auto insort
RRL-> ADD
New constant set is: { E, 0 }
Equations read in are:
 1. MAX(X, 0) == X  [USER, 1]
 2. MAX(0, X) == X  [USER, 2]
 3. MAX(S(X), S(Y)) == S(MAX(X, Y))  [USER, 3]
 4. MIN(X, 0) == 0  [USER, 4]
 5. MIN(0, X) == 0  [USER, 5]
 6. MIN(S(X), S(Y)) == S(MIN(X, Y))  [USER, 6]
 7. SORT(E) == E  [USER, 7]
 8. SORT((X + Y)) == INSERT(X, SORT(Y))  [USER, 8]
 9. INSERT(X, E) == (X + E)  [USER, 9]
10. INSERT(X, (Y + Z)) == (MIN(X, Y) + INSERT(MAX(X, Y), Z))  [USER, 10]


Type Add, Akb, Auto, Break, Clean, Delete, Grammar, History, Init, Kb, List,
     Load, Log, Makerule, Modal, Narrow, Norm, Option, Operator, Prove, Quit,
     Read, Refute, Save, Solve, Stats, Suffice, Undo, Unlog, Write or Help.
RRL-> OPERATOR

PRECEDENCE
SORT INSERT MAX MIN S +
Precedence relation, SORT > INSERT, is added.
Precedence relation, INSERT > MAX, is added.
Precedence relation, MAX > MIN, is added.
Precedence relation, MIN > S, is added.
Precedence relation, S > +, is added.


Type Add, Akb, Auto, Break, Clean, Delete, Grammar, History, Init, Kb, List,
```

```
     Load, Log, Makerule, Modal, Narrow, Norm, Option, Operator, Prove, Quit,
     Read, Refute, Save, Solve, Stats, Suffice, Undo, Unlog, Write or Help.
RRL-> kb
Your system is locally confluent.

  [1] MAX(X, 0) ---> X  [USER, 1]
  [2] MAX(0, X) ---> X  [USER, 2]
  [3] MAX(S(X), S(Y)) ---> S(MAX(X, Y))  [USER, 3]
  [4] MIN(X, 0) ---> 0  [USER, 4]
  [5] MIN(0, X) ---> 0  [USER, 5]
  [6] MIN(S(X), S(Y)) ---> S(MIN(X, Y))  [USER, 6]
  [7] SORT(E) ---> E  [USER, 7]
  [8] SORT((X + Y)) ---> INSERT(X, SORT(Y))  [USER, 8]
  [9] INSERT(X, E) ---> (X + E)  [USER, 9]
 [10] INSERT(X, (Y + Z)) ---> (MIN(X, Y) + INSERT(MAX(X, Y), Z))  [USER, 10]

Time used                           =   0.25 sec
Number of rules generated           = 10
Number of rules retained            = 10
Number of critical pairs            = 1
Time spent in normalization         =   0.00 sec  (0.0 percent of time)
Time spent in unification           =   0.02 sec   (6.0 percent of time)
Time spent in ordering              =   0.07 sec   (26.0 percent of time)
Time spent in simplifying the rules  =   0.03 sec   (13.0 percent of time)
Total time (including 'undo' action) = 16.0 sec



Type Add, Akb, Auto, Break, Clean, Delete, Grammar, History, Init, Kb, List,
     Load, Log, Makerule, Modal, Narrow, Norm, Option, Operator, Prove, Quit,
     Read, Refute, Save, Solve, Stats, Suffice, Undo, Unlog, Write or Help.
```

```
RRL-> sol

 Give the equation you wish to solve. Use  'option solve' from RRL's
    top-level for help and setting options.
Type your equation in the format:  L == R (if C)
Enter a ']' to exit when no equation is given.
sort(X) ==    (0 + (S(0) + (S(S(0)) + E)))
 Solving SORT(X) == (0 + (S(0) + (S(S(0)) + E)))  [USER, 11]
   X  |--> (+ (S (S (0))) (+ (S (0)) (+ (0) (E))))
Found at Depth 16.
Try Again?(Y,N) y
   X  |--> (+ (S (S (0))) (+ (0) (+ (S (0)) (E))))
Found at Depth 16.
Try Again?(Y,N) y
   X  |--> (+ (S (0)) (+ (S (S (0))) (+ (0) (E))))
Found at Depth 16.
Try Again?(Y,N) y
   X  |--> (+ (S (0)) (+ (0) (+ (S (S (0))) (E))))
Found at Depth 16.
Try Again?(Y,N) y
/*************************************************************************/
/* The system cannot exhaust all solutions within given depth bound,  and */
/* therefore has to increase the depth-bound to complete the search. Thus */
/* it generates some of the initial solutions a second time.              */
/*************************************************************************/
   X  |--> (+ (S (S (0))) (+ (S (0)) (+ (0) (E))))
Found at Depth 16.
Try Again?(Y,N) y
   X  |--> (+ (S (S (0))) (+ (0) (+ (S (0)) (E))))
Found at Depth 16.
Try Again?(Y,N) y
```

```
   X  |--> (+ (S (0)) (+ (S (S (0))) (+ (0) (E))))
Found at Depth 16.
Try Again?(Y,N) y
   X  |--> (+ (S (0)) (+ (0) (+ (S (S (0))) (E))))
Found at Depth 16.
Try Again?(Y,N) y
   X  |--> (+ (0) (+ (S (S (0))) (+ (S (0)) (E))))
Found at Depth 17.
Try Again?(Y,N) y
   X  |--> (+ (0) (+ (S (0)) (+ (S (S (0))) (E))))
Found at Depth 17.
Try Again?(Y,N) y
No more solutions to this equation.

 Total Time used for solving =    29.65 seconds


Type Add, Akb, Auto, Break, Clean, Delete, Grammar, History, Init, Kb, List,
     Load, Log, Makerule, Modal, Narrow, Norm, Option, Operator, Prove, Quit,
     Read, Refute, Save, Solve, Stats, Suffice, Undo, Unlog, Write or Help.
RRL-> quit
```

BIBLIOGRAPHY

[Arnborg and Tidén, 1985] A. Arnborg and E. Tidén. Unification problems with one-sided distributivity. In *Proceedings of the First International Conference on Rewriting Techniques and Applications*. Volume 202, pages 398–406, of *Lecture Notes in Computer Science*, Springer-Verlag, 1985.

[Baader, 1986] A. Baader. Unification in idempotent semigroups is of type zero. *Journal of Automated Reasoning*, Volume 2, Number 3, pages 283–286, 1986.

[Baader and Büttner, 1988] F. Baader and W. Büttner. Unification in commutative, idempotent monoids. *Theoretical Computer Science*, Volume 56, Number 1, pages 345–352, 1988.

[Baader and Siekmann, 1993] F. Baader and J. H. Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press, 1993.

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

[Bachmair, 1992] L. Bachmair. Associative-commutative reduction orderings. *Information Processing Letters*, Volume 43, pages 21–27, 1992.

[Bachmair and Plaisted, 1985] L. Bachmair and D. A. Plaisted. Termination orderings for associative-commutative rewrite systems. *Journal of Symbolic Computation*, Volume 1, pages 329–349, 1985.

[Barbuti *et al.*, 1986] R. Barbuti, M. Bellia, G. Levi and M. Martelli. LEAF: a language which integrates logic, equations and functions. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 201–238, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Ben Cherifa and Lescanne, 1987] A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, Volume 9, pages 137–159, 1987.

[Bellia and Levi, 1986] M. Bellia and G. Levi. The relation between logic and functional languages: a survey. *Journal of Logic Programming*, Volume 3, Number3, pages 217–236, 1986.

[Björner, 1982] D. Björner, editor. Proceedings of the IFIP working conference on formal description of programming concepts–II. Garmisch-Partenkirchen, West Germany, North-Holland, 1982.

[Bockmayr, 1987] A. Bockmayr. A note on a canonical theory with undecidable unification and matching problem. *Journal of Automated Reasoning*, Volume 3, pages 379–381, 1987.

[Bockmayr *et al.*, 1992] A. Bockmayr, S. Krischer and A. Werner. An optimal narrowing strategy for general canonical systems. In *Proceedings of the Third International Workshop on Conditional Term Rewriting Systems*, France, 1992. Volume 656, pages 483–497, of *Lecture Notes in Computer Science*, Springer Verlag.

[Bosco *et al.*, 1987] P. G. Bosco, E. Giovanneti and C. Moiso. Refined strategies for semantic unification. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development.* Volume 250, pages 276-290, of *Lecture Notes in Computer Science*, Springer Verlag, 1987.

[Breazu-Tannen, 1988] V. Breazu-Tannen. Combining algebra and higher-order types. In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science*, pages 82–90, 1988.

[Chabin and Réty, 1991] J. Chabin and P. Réty. Narrowing directed by a graph of terms. In *Proceedings of the Fourth International Conference on Rewriting Techniques and Applications*, Como, Italy, 1991. Volume 488, pages 112–123, of *Lecture Notes in Computer Science*, Springer Verlag.

[Cheong and Fribourg, 1993] P. H. Cheong and L. Fribourg. Implementation of narrowing: the prolog-based approach. In K. R. Apt, J. W. de Bakker, and J. Rutten, editors, *Logic*

Programming Languages, Constraints, Functions and Objects, chapter 1, pages 1–20, MIT Press, 1993.

[Christian, 1992] J. Christian. Some termination criteria for narrowing and E-narrowing. In *Proceeding of the Eleventh International Conference on Automated Deduction*, Saratoga Springs, New York, 1992. Volume 607, pages 582–588, of *Lecture Notes in Artificial Intelligence*, Springer Verlag.

[Clocksin and Mellish, 1981] W. F. Clocksin and C. S. Mellish. Programming in Prolog. Springer Verlag, 1981.

[Comon, 1992] H. Comon. On unification of terms with integer exponents. Technical Report 770, Universite de Paris-Sud, Laboratoire de Recherche en Informatique, 1992.

[Comon *et al.*, 1991] H. Comon, M. Haberstrau and J.-P. Jouannaud. Decidable problems in shallow equational theories. Technical Report 718, Universite de Paris-Sud, Laboratoire de Recherche en Informatique, 1991.

[Contejean, 1992] E. Contejean. A partial solution for D-unification based on a reduction to AC1-unification. Journal of Symbolic Computation, to appear.

[DeGroot and Lindstrom, 1986] D. DeGroot and G. Lindstrom, editors. Logic programming: Functions, Relations and Equations. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Delor and Puel, 1993] C. Delor and L. Puel. Extension of the associative path ordering to a chain of associative-commutative symbols. In *Proceedings of the Fifth International Conference on Rewriting Techniques and Applications*, Montreal, Canada, 1993. Volume 690, pages 389–404, of *Lecture Notes in Computer Science*, Springer Verlag.

[Dershowitz, 1982] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, Volume 17, pages 279–301, 1982.

[Dershowitz, 1987] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, Volume 3, pages 69–116, 1987.

[Dershowitz and Josephson, 1984] N. Dershowitz and A. N. Josephson. Logic programming by completion. Proceedings of the Second International Logic Programming Conference, Uppsala, Sweden, pages 313–320, 1984.

[Dershowitz *et al.*, 1983] N. Dershowitz, J. Hsiang, N. A. Josephson and D. A. Plaisted. Associative-commutative rewriting. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence,* Karlsruhe, West Germany, pages 940–944, 1983.

[Dershowitz and Jouannaud, 1990] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 243–320, North-Holland, Amsterdam, 1990.

[Dershowitz and Manna, 1979] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, Volume 22, pages 465–476, 1979.

[Dershowitz and Mitra, 1992] N. Dershowitz and S. Mitra. Path orderings for termination of associative-commutative rewriting. In *Proceedings of the Third International Workshop on Conditional Term Rewriting Systems*, France, 1992. Volume 656, pages 168–174, of *Lecture Notes in Computer Science*, Springer Verlag.

[Dershowitz and Mitra, 1993] N. Dershowitz and S. Mitra. Higher-order and semantic unification. In *Proceedings of the Thirteenth International Conference on Foundations of Software Technology and Theoretical Computer Science*, Bombay, India, 1993. Volume 761, pages 139–150, of *Lecture Notes in Computer Science*, Springer Verlag.

[Dershowitz *et al.*, 1992] N. Dershowitz, S. Mitra and G. Sivakumar. Decidable matching for convergent systems. In *Proceedings of the Eleventh Conference on Automated Deduction*, 1992. Volume 607, pages 589–602, of *Lecture Notes in Artificial Intelligence*, Springer Verlag.

[Dershowitz *et al.*, 1990] N. Dershowitz, S. Mitra and G. Sivakumar. Equation solving in conditional AC-theories. In *Proceedings of the Second International Conference on Algebraic and Logic Programming*, Nancy, France, 1990. Volume 463, pages 283–297, of *Lecture Notes in Computer Science*, Springer Verlag.

[Dershowitz and Okada, 1990] N. Dershowitz and M. Okada. A rationale for conditional equational programming. *Theoretical Computer Science*, Volume 75, pages 111–138, 1990.

[Dershowitz *et al.*, 1987] N. Dershowitz, M. Okada and G. Sivakumar. Confluence of conditional rewrite systems. In *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, Orsay, France, 1987. Volume 308, pages 31–44, of *Lecture Notes in Computer Science*, Springer Verlag.

[Dershowitz *et al.*, 1988] N. Dershowitz, M. Okada and G. Sivakumar. Canonical conditional rewrite systems. In *Proceedings of the Ninth Conference on Automated Deduction*, Argonne, IL, 1988. Volume 310, pages 538–549, of *Lecture Notes in Computer Science*, Springer Verlag.

[Dershowitz and Plaisted, 1988] N. Dershowitz and D. A. Plaisted. Equational programming. In J. E. Hayes, D. Michie and J. Richards, editors, *Machine Intelligence 11: The logic and acquisition of knowledge*, chapter 2, pages 21–56, Oxford Press, Oxford, 1988.

[Dershowitz and Sivakumar, 1987] N. Dershowitz and G. Sivakumar. Solving goals in equational languages. In *Proceedings of the First International Workshop Conditional Term Rewriting System*, Orsay, France, 1987. Volume 308, pages 45–55, of *Lecture Notes in Computer Science*, Springer Verlag.

[Dershowitz and Sivakumar, 1988] N. Dershowitz and G. Sivakumar. Goal-directed equation solving. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, St. Paul, MN, pages 166–170, 1988.

[Dougherty, 1991] D. J. Dougherty. Adding algebra to the untyped lambda-calculus. In *Proceedings of the Fourth International Conference on Rewriting Techniques and Applications*, Como, Italy, 1991. Volume 488, pages 37–48, of *Lecture Notes in Computer Science*, Springer Verlag.

[Dougherty, 1993] D. J. Dougherty. Higher-order unification via combinators. *Theoretical Computer Science*, Volume 114, pages 273–298, 1993.

[Dougherty and Johann, 1990] D. J. Dougherty and P. Johann. An improved general E-unification method. In *Proceedings of the Tenth Conference on Automated Deduction*,

1990. Volume 449, pages 261–275, of *Lecture Notes in Artificial Intelligence*, Springer Verlag.

[Dougherty and Johann, 1992] D. J. Dougherty and P. Johann. A combinatory logic approach to higher-order E-unification. In *Proceedings of the Eleventh Conference on Automated Deduction*, Saratoga Springs, New York, 1992. Volume 607, pages 79–93, of *Lecture Notes in Computer Science*, Springer Verlag.

[Dowek, 1992] G. Dowek. Third-order matching is decidable. In *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science*, pages 1–10, Santa Cruz, California, 1992.

[Fages, 1984] F. Fages. Associative-commutative unification. In *Proceedings of the Seventh International Conference on Automated Deduction*, 1984. Volume 170, pages 194–208, of *Lecture Notes in Computer Science*, Springer Verlag.

[Fay, 1979] M. Fay. First-order unification in an equational theory. In *Proceedings of the Fourth Workshop on Automated Deduction*, pages 161–167, Austin, TX, 1979.

[Fribourg, 1985] L. Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 172–184, Boston, MA, 1985.

[Gallier and Snyder, 1989] J. H. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, Volume 67, pages 203-260, North-Holland, 1989.

[Goguen and Meseguer, 1984] J. A. Goguen and J. Meseguer. Equality, types, modules and generics for logic programming. In *Proc. of the Second International Logic Programming Conference*, pages 115–125, 1984.

[Hanus, 1993] M. Hanus. Presonal Communication, summer 1993.

[Heilbrunner and Hölldobler, 1987] S. Heilbrunner and S. Hölldobler. The undecidability of the unification and matching problem for canonical theories. *Acta Informatica*, Volume 24, pages 157–171, 1987.

[Henderson, 1980] P. Henderson. Functional programming: application and implementation. Prentice-Hall International Series in Computer Science, 1980.

[Hermann, 1992] M. Hermann. On the relation between primitive recursion, schematization and divergence. In *Proceedings of the Third International Conference on Algebraic and Logic Programming*, Volterra, Italy, 1992. Volume 632, pages 115–127, of *Lecture Notes in Computer Science*, Springer Verlag.

[Hölldobler, 1987] S. Hölldobler. A unification algorithm for confluent theories. In *Proceedings of the Fourteenth International Conference on Automata, Languages and Programming*, 1987. Volume 267, pages 31–41, of *Lecture Notes in Computer Science*, Springer Verlag.

[Hopcroft and Ullman, 1979] J. E. Hopcroft and J. D. Ullman. Introduction to automata theory, languages and computation. Addison Wesley, 1979.

[Hsiang and Jouannaud, 1988] J. Hsiang and J.-P. Jouannaud. General E-unification revisited. In *Proceedings of the Second International Workshop on Unification*, 1988.

[Huet, 1975] G. P. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, Volume 1, Pages 27–57, 1975.

[Huet and Oppen, 1980] G. Huet and D. C. Oppen. Equations and rewrite rules: a survey. In, R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405, Academic Press, New York, 1980.

[Hullot, 1980] J.-M. Hullot. Canonical forms and unification. In *Proceedings of the Fifth International Conference on Automated Deduction*, Les Arcs, France, 1980. Volume 87, pages 318–334, of *Lecture Notes in Computer Science*, Springer Verlag.

[Josephson and Dershowitz, 1989] N. A. Josephson and N. Dershowitz. An implementation of narrowing. *Journal of Logic Programming*, Volume 6 (1&2), pages 57–77, 1989.

[Jouannaud *et al.*, 1983] J.-P. Jouannaud, C. Kirchner and H. Kirchner. Incremental construction of unification algorithms in equational theories. In *Proceedings of the International Colloquium on Automata, Languages and Programming*, 1983. Volume 154, pages 361–373, of *Lecture Notes in Computer Science*, Springer Verlag.

[Jouannaud and Kirchner, 1991] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, MIT Press, Cambridge, MA, 1991.

[Kamin, 1990] S. N. Kamin. Programming languages: an interpreter-based approach. Addison-Wesley, 1990.

[Kapur and Narendran, 1987] D. Kapur and P. Narendran. Matching, unification and complexity. *ACM SIGSAM Bulletin*, Volume 21, Number 4, pages 6–9, 1987.

[Kapur *et al.*, 1990] D. Kapur, G. Sivakumar and H. Zhang. A new method for proving termination of AC-rewrite systems. In *Proceedings of the Tenth International Conference of Foundations of Software Technology and Theoretical Computer Science*, 1990. Volume 472, pages 133–148, of *Lecture Notes in Computer Science*, Springer Verlag.

[Kapur and Zhang, 1987] D. Kapur and H. Zhang. RRL: a rewrite rule laboratory, user's manual, 1987.

[Kirchner, 1984] C. Kirchner. A new equational unification method: a generalization of Martelli-Montanari's algorithm. In *Proceedings of the Seventh International Conference on Automated Deduction*, 1984.

[Kirchner, 1986] C. Kirchner. Computing unification algorithms. In *Proceedings of the first IEEE Symposium on Logic in Computer Science*, pages 206–216, 1986.

[Kirchner, 1989] C. Kirchner. From unification in combination of equational theories to a new AC-unification algorithm. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, Volume 2, pages 171–210, Academic Press, New York, 1989.

[Klop, 1992] J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, Volume 2, pages 1–117, Oxford University Press, Oxford, 1992.

[Kowalski, 1979] R. A. Kowalski. Logic for problem solving. North Holland Publishing Company, 1979.

[Lankford, 1979] D. S. Lankford. On proving term rewriting systems are Noetherian. Memo MTP-3, Mathematics Department, Louisiana Tech. University, Ruston, LA, 1979.

[Lescanne, 1990] P. Lescanne. On the recursive decomposition ordering with lexicographic status and other related orderings. *Journal of Automated Reasoning*, Volume 6, pages 39–49, 1990.

[Lindstrom, 1985] G. Lindstrom. Functional programming and the logical variable. In *Proc. of the ACM Symposium on Principles of Programming Languages*, 1985.

[Makanin, 1977] G. S. Makanin. The problem of solvability of equations in a free semigroup, 1977.

[Martelli and Montanari, 1982] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, Volume 4, Number 2, pages 258–282, 1982.

[Martelli *et al.*, 1989] A. Martelli, G. F. Rossi and C. Moiso. Lazy unification algorithms for canonical rewrite systems. In H. Ait-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, pages 245–274, Academic Press, New York, 1989.

[Middeldorp and Hamoen, 1992] A. Middledorp and E. Hamoen. Counterexamples to completeness results for basic narrowing. In *Proceedings of the Third International Conference on Algebraic and Logic Programming*, Volterra, Italy, 1992. Volume 632, pages 244–258, of *Lecture Notes in Computer Science*, Springer Verlag.

[Mitra, 1990] S. Mitra. Top-down equation solving and extensions to associative and commutative theories. Master's thesis, Department of Computer and Information Sciences, University of Delaware, Newark, DE, 1990.

[Mitra and Sivakumar, 1991] S. Mitra and G. Sivakumar. AC-equation solving. In *Proceedings of the Eleventh International Conference on Foundations of Software Technology and Theoretical Computer Science*, New Delhi, India, 1991. Volume 560, pages 40–56, of *Lecture Notes in Computer Science*, Springer Verlag.

[Nipkow and Qian, 1991] T. Nipkow and Z. Qian. Modular higher-order E-unification. In *Proceedings of the Fourth International Conference on Rewriting Techniques and Applications*, Como, Italy, 1991. Volume 488, pages 200–214, of *Lecture Notes in Computer Science*, Springer Verlag.

[Nutt *et al.*, 1989] W. Nutt, P. Réty and G. Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, Volume 7, pages 295–317, 1989.

[Paulson, 1991] L. C. Paulson. ML for the working programmer. Cambridge University Press, 1991.

[Paterson and Wegman, 1978] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, Volume 16, Number 2, pages 158–167, 1978.

[Plotkin, 1972] G. Plotkin. Building in equational theories. *Machine Intelligence*, Volume 7, pages 73–90, 1972.

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

[Réty, 1987] P. Réty. Improving basic narrowing techniques. In *Proceedings of the Second International Conference on Rewriting Techniques and Applications*, Bordeaux, France, 1987. Volume 256, pages 228–241, of *Lecture Notes in Computer Science*, Springer Verlag.

[Robinson, 1965] J. A. Robinson. A machine oriented logic based on the resolution principle. *Journal of the ACM*, Volume 12, Number 1, pages 23–41, 1965.

[Robinson and Wos, 1969] G. A. Robinson and L. Wos. Paramodulation and theorem proving in first order theories with equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, Volume 4, American Elsevier, New York, pages 135–150, 1969.

[Rubio and Nieuwenhuis, 1993] A. Rubio and R. Nieuwenhuis. A precedence-based total AC-compatible ordering. In *Proceedings of the Fifth International Conference on Rewriting Techniques and Applications*, Montreal, Canada, 1993. Volume 690, pages 374–388, of *Lecture Notes in Computer Science*, Springer Verlag.

[Siekmann, 1979] J. H. Siekmann. Unification of commutative terms. In *Proceedings of the Conference on Symbolic and Algebraic Manipulation*, 1979. Volume 72, pages 531–545, of *Lecture Notes in Computer Science*, Springer Verlag.

[Siekmann, 1989] J. H. Siekmann. Unification theory. *Journal of Symbolic Computation*, Volume 7 (3 & 4), pages 207–274, 1989.

[Siekmann and Szabó, 1984] J. H. Siekmann and P. Szabó. Universal unification and classification of equational theories. In *Proceedings of the Seventh International Conference on Automated Deduction*, 1984. Volume 170, pages 1–42, of *Lecture Notes in Computer Science*, Springer Verlag.

[Sivakumar, 1989] G. Sivakumar. Proofs and computations in conditional equational theories. PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1989.

[Snyder, 1990] W. Snyder. Higher-order E-unification. In *Proceedings of the Tenth International Conference on Automated Deduction*, Kaiserslautern, FRG, 1990. Volume 449, pages 573–587, of *Lecture Notes in Computer Science*, Springer Verlag.

[Snyder, 1991] W. Snyder. A proof theory for general E-unification. Volume 11, Progress in Computer Science and Applied Logic, Birkhäuser, 1991.

[Snyder and Gallier, 1989] W. Snyder and J. Gallier. Higher-order unification revisited: complete sets of transformations. *Journal of Symbolic Computation*, Volume 8, pages 101–140, 1989.

[Socher-Ambrosius, 1993] R. Socher-Ambrosius. Unification of terms with exponents. Technical Report, MPI-I-93-217, Max-Plank Institute fur Informatik, 1993.

[Stickel, 1981] M. E. Stickel. A unification algorithm for associative-commutative functions. *JACM*, Volume 28, pages 423–434, 1981.

[Subrahmanyam and You, 1986] P. A. Subrahmanyam and J.-H. You. FUNLOG: a computational model integrating logic programming and functional programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 157–200, Prentice-Hall, Englewood Cliffs, NJ, 1986.

[You, 1989] J.-H. You. Enumerating outer narrowing derivations for constructor-based term rewriting systems. *Journal of Symbolic Computation*, Volume 7, pages 319–341, 1989.

VITA

Subrata Mitra was born on September 29, 1965 in Calcutta, India. He attended the Indian Institute of Technology, Kanpur, India, and received his Bachelor of Technology degree in Computer Science and Engineering, in May 1988.

Subrata attended the University of Delaware between September, 1988, and December 1990, and received the Master of Science degree in Computer and Information Sciences, in January 1991. Thereafter, Subrata joined the University of Illinois at Urbana-Champaign for his doctoral studies. On completion of his Ph.D., Subrata has joined the I.B.M. Corporation as a Development Staff Member at the Santa Teresa Laboratories.