Equational Reasoning and Term Rewriting Systems

David A. Plaisted

Contents

(From the Handbook of Logic in Artificial Intelligence and Logic Programming, volume 1, edited by Dov Gabbay and Jorg Siekmann, newly published by Oxford University Press.)

An *equational system* is a set of equations. Often we are interested in knowing if an equation follows logically from the given set. For example, given the equations $x + y = y + x$, $(x + y) + z = x + (y + z)$, and $-(-(x + y) + -(x + -y)) = x$, we might want to know if the equation $-(-x + y) + -(-x + -y) = x$ is a logical consequence. As another example, we might want to know if $x * y = y * x$ in a group in which $x^2 = e$ for all x. Such systems are of interest for computer scientists as well as mathematicians. Common data structures like lists and stacks can often be described by such sets of equations. Furthermore, systems for mechanising such proofs on a computer are becoming more and more powerful. In addition, a functional program is essentially a set of equations, typically with higher order functions, and the execution of a program is then a kind of equational reasoning. We will discuss methods of inference that are particularly adapted to equational systems without explicit higher order functions. We will also discuss systems in which the equations may have conditions attached. Next we will consider the use of equational reasoning in general theorem proving programs. We also touch on the use of equations in logic and functional programming. For other surveys, see [Klo91, DJ90]. Our access to copies of these surveys has made this writing task considerably easier, and we acknowledge our debt with gratitude. Our survey differs from these in several ways. In general, our survey uses formalisms based on logic, and emphasizes the relationship between syntax and semantics. Some of our notation is novel, and some results may be original. Our presentation of some topics like unfailing completion and strong sequentiality is different. We do not emphasize properties of systems with more than one rewrite relation, as does [Klo91]. We do not study inductive theorems; for that, see chapter **??**. The organization of our survey is also different than others.

1 Syntax of equational systems

We use $f, g, h, \ldots$ as *function symbols*, $a, b, c, \ldots$ as *individual constants*, $x, y, z, \ldots$ as *individual variables*, and $=$ as the equality symbol. Each function symbol has an *arity*, which is a non-negative integer telling how many arguments it takes. A *term* is either a variable, an individual constant, or an expression of the form $f(t_1, t_2, \ldots, t_n)$ where f is a function symbol of arity n and the t_i are terms. We use $r, s, t, \ldots$ for terms. We say a term u is a *subterm* of t if u is t or if t is $f(t_1, \ldots, t_n)$ and u is a subterm of t_i for some i. An *equation* is an expression of the form $s = t$ where s and t are terms. An *equational system* is a set (often represented by a list) of equations. We will generally consider only unsorted equational systems, for simplicity, while occasionally referring to the sorted case. We use E to refer to equational systems.

2 Semantics of equational systems

We define the semantics of equational systems in the style of the semantics of first-order logic. An algebraic approach is also possible, but we do not use it in order to reduce the mathematical machinery needed, and also to emphasize connections with logic. The algebraic approach essentially gives a convenient formalism for describing structures, which we now define. A *structure* M consists of a non-empty domain D and an assignment of a meaning f^M to each function symbol f (including individual constants). A function symbol of arity n is assigned a meaning $f^M : D^n \rightarrow D$, that is, a function which maps n-tuples of elements of D into D. Also, M includes an assignment v of meanings to the variables. A variable x is assigned a meaning $v(x)$ which is an element of D. The meaning t^M of a term t is defined as follows: If t is a variable x then t^M is $v(x)$. If t is of the form $f(t_1, t_2, ..., t_n)$ then t^M is $f^M(t_1{}^M, t_2{}^M, ..., t_n{}^M)$. If M is a structure, we write $M \models s = t$ to indicate that the equation $s = t$ is true in M, or more formally, that M *satisfies* the equation $s = t$. This is defined as follows: $M \models s = t$ iff s^M and t^M are identical elements of D, that is, s and t are given the same meaning by M. We say that a structure M satisfies a set E of equations, written $M \models E$, if M satisfies each element of E. Such a structure is called a *model* of E. If E_1 and E_2 are two equational systems, we write $E_1 \models E_2$ if all structures M that satisfy E_1 also satisfy E_2, that is, all models of E_1 are also models of E_2. In this case, if $E_1 \models E_2$, we call E_2 a *logical consequence* of E_1.

The question of interest for us is to determine when $E_1 \models E_2$ for various E_1 and E_2. Usually E_2 will consist of a single equation, in which case we are interested in knowing whether this equation is a logical consequence of E_1. Intuitively, this means that the equation is necessarily true whenever all the equations in E_1 are true. In order to determine this, many proof systems have been developed, and recently many of them have been adapted so they can be implemented on computers.

Sometimes we are interested in equations with existential quantifiers. We may want to know if $E \models ((\exists x)s = t)$, that is, from E does it follow that there is an x such that $s = t$? We formally define the semantics of $\exists$ as follows: $M \models (\exists x A)$ if there is a structure M' similar to M except for the assignment to x, such that $M' \models A$.

3 Sorts

We briefly indicate how the syntax and semantics of equational systems differ in the sorted case. Our presentation is close to that of [Gal86]. For this, we assume that there is a countable set S of *sorts*. In the above case, there was only one sort. In programming languages, sorts correspond to the different kinds of objects that may be considered, like lists, Booleans, integers, reals, and so on. Some functions only take arguments of some sorts and return results of other sorts. For example, in list processing, it is not correct to evaluate $car(x)$ where x is an integer. Sorted logic attempts to capture this intuition. Also, it sometimes permits more efficient inference, since sets of objects that appear in certain problems (like animals, birds, plants) can be specified as sorts. We assume that there is a *rank function* f, which for each function symbol f assigns a *rank* $r(f)$ which is a pair (u, s) where u is a sequence of sorts and s is a sort. The first component u is the arity of f, and s is the sort (or type) of f. The intuition is that the arguments of f have specified sorts, so that if the arity is s_1, s_2 then the first argument of f should have sort s_1 and the second argument should have sort s_2, that is, the terms should be *well-sorted.* Thus some terms are not well sorted and are not considered as well-formed terms in the sorted case. In the unsorted case, the arity is just an integer telling the number of arguments. The second component s of $r(f)$ is the sort of the value returned by f, that is, the sort of $f(t_1, t_2)$ where the arguments t_i have appropriate sorts. Also, each variable x is assigned a rank (e, s) for some sort s, where e is the empty sequence, indicating that the variable does not take arguments like a function. Note that individual constants also have ranks of the form (e, s) for the same reason. We abbreviate a rank of the form $((s_1, s_2, ..., s_n), s)$ by $(s_1, s_2, ..., s_n, s)$. Here the arity is $s_1, s_2, ..., s_n$ and the sort is s.

The semantics for sorts are defined in terms of a *many sorted structure* M which consists of a collection of *carriers* M_s, one for each sort s. These are non-empty, analogous to the domain D above, but there may be more than one since there may be more than one sort s in S. Also, a structure specifies a meaning f^M for function symbols f and an assignment v for variables. If f is of rank $(s_1, s_2, ..., s_n, s)$ then f^M is a function from $M_{s_1} \times M_{s_2} \times ... \times M_{s_n}$ to the carrier M_s. If x is of rank (e, s) then $v(x)$ is an element of the carrier M_s. The semantics t^M of a term t of the form $f(t_1, t_2, ..., t_n)$ is $f^M({t_1}^M, {t_2}^M, ..., {t_n}^M)$, as before. The notion $M \models E$ of satisfaction is defined as above, and the notion $E_1 \models E_2$ of logical consequence is also as before.

This can be extended to the *order sorted* case. The idea there is that there can be a partial order on the sorts. If $s_1 < s_2$ then s_1 is a *subsort* of s_2, which means that its carrier is a subset of the carrier for s_2. This models many naturally occurring situations; for example, birds are a subset of animals. Another extension is to *polymorphism*. It is often unrealistic to restrict a function to have only one rank. For example, a function that takes the length of a list may work equally well on lists of integers, lists of lists, lists of Booleans, et cetera. These may all be different sorts. Therefore the sort of such a function is defined polymorphically, which means that the rank has a certain amount of flexibility.

4 Solving for values

Sometimes we want to know whether there exists a value having certain properties. For example, given a reasonable definition of the append function, which concatenates two lists, we may want to know whether there exists an x such that $append(x, [3, 4]) = [1, 2, 3, 4]$. We can define append by $append([], x) = x$ and $append([x|y], z) = [x|append(y, z)]$, using $[x|y]$ for $cons(x, y)$ and $[1, 2, 3, 4]$ for $cons(1, cons(2, cons(3, cons(4, []))))$, et cetera. Letting E be the set of defining equations for *append*, we are asking whether $E \models (\exists x)\ append(x, [3, 4]) = [1, 2, 3, 4]$. Usually we also want to know which x satisfies this equation. In this case, of course, $x = [1, 2]$ works since if we concatenate the lists $[1, 2]$ and $[3, 4]$ we obtain $[1, 2, 3, 4]$. Solving for x is similar to finding a constructive proof of the theorem $(\exists x)\ append(x, [3, 4]) = [1, 2, 3, 4]$. The technique of *narrowing* [Fay79], discussed in section 17, is used to solve such problems. This is an interesting application for programming since the variable x can be viewed as an output of the program and so equational logic can be used as a programming language in this way. Note that the following query is syntactically similar: $E \models (\exists x) append([1, 2], [3, 4]) = x$. However, queries of this form are usually much easier to solve, it turns out. (The answer $x = append([1, 2], [3, 4])$ is not typically what is wanted, though it is formally correct!) The idea of solving for variables satisfying equations is also the basis for E-unification [Plo72]; we will briefly touch on this later, and it is more fully discussed in chapter **??** in this volume.

There is also some interest in solving inequations, that is, finding a value making the equation $s = t$ false. For example, see [CL89], where this problem is discussed, and see [Bur88], where an application to AC unification [Sti81] is given.

5 Inductive theorems

There is a subtlety in the notion of logical consequence which should be mentioned here. The structures M may include elements that are inaccessible, that is, which cannot be represented by any finite variable-free term. (A term or formula without variables is called *ground*.) For example, if E includes the constant 0 and the unary function symbol s, then the ground terms are $\{0, s(0), s(s(0)), ...\}$. However, the domain D of the structure M may include elements d that cannot be represented by any of these elements; formally, it may be that $d \neq 0^M, d \neq s(0)^M, d \neq s(s(0))^M, ...$ This means that the notion of logical consequence may not agree with our intuition. Often we think of logical consequences as those equations $s = t$ such that whenever the variables of s and t are replaced by ground terms in a systematic way, the result is a consequence of E. We define a *ground instance* of an equation $s = t$ to be an equation $s' = t'$ in which all variables are replaced by ground terms in a consistent way. Thus a ground instance of $f(x, y) = x$ is $f(g(a), h(b)) = g(a)$. So, we are often interested in those equations $s = t$ such that all ground instances are logical consequences of E. For example, suppose E is $\{x + 0 = x, x + s(y) = s(x + y)\}$, writing $+$ as an infix binary function symbol. These equations completely define the usual addition operation on non-negative integers, regarding s as the successor function. We expect addition to be commutative, which means that we expect to have $E \models (u+v = v+u)$. Indeed, any ground instance of this equation containing only the function symbols s and 0 (such as the instance $s(0) + s(s(0)) = s(s(0)) + s(0)$) is in fact a logical consequence of E. However, the equation $u + v = v + u$ is *not* a logical consequence of E. It is possible to construct a structure M making E true and the equation $u + v = v + u$ false. This structure contains inaccessible elements, as mentioned above.

In fact, by Godel's incompleteness theorem, there can be no method of deriving all the equations true on the nonnegative integers. This means that every set W of axioms for the integers has a nonstandard model, in which some theorem A that is true on the integers is false. This model contains inaccessible elements. Another way of looking at this is that it is impossible in a finite axiom system to fully characterize the integers, or the set of finite terms. Any predicate that is true of all finite ground terms, will also be true on some infinite terms (inaccessible elements), in some structure. We hope that this informal treatment will give some idea of the logical subtleties involved.

However, there is still some interest in proving theorems that are true on the standard integers, finite terms, etc. Formally, the *inductive theory* of a set E of equations is the set of equations $s = t$ such that all ground instances of $s = t$ are logical consequences of E. There are some methods for deriving and verifying equations in the inductive theory. These involve some form of mathematical induction. For a discussion of this, see chapter **??**.

6 Proof systems

We now discuss methods for showing that $E \models s = t$ for various E, s, and t.

Definition 6.0.1. A *substitution* is a function from variables to terms, which is the identity on all but finitely many variables.

We typically use Greek letters such as θ, ρ, σ for substitutions. If θ is a substitution and x is a variable, then we indicate the application of θ to x by $\theta(x)$ as usual. We also extend substitutions homomorphically to terms. If t is a term and θ is a substitution, we indicate by $t\theta$ the application of θ to t. This is defined inductively as follows: If t is a variable then $t\theta$ is $\theta(t)$. Otherwise, t is of the form $f(t_1, t_2, ..., t_n)$ for some n (possibly zero). Then $t\theta$ is $f(t_1\theta, t_2\theta, ..., t_n\theta)$. Thus $t\theta$ is t with all variables replaced by terms as specified by θ. We often write a substitution as a set $\{x_1 \mapsto s_1 \ldots x_n \mapsto s_n\}$, indicating that the variable x_i is mapped to s_i by the substitution. We call $t\theta$ an *instance* of t. Also, if $t = u$ is an equation and θ is a substitution, we call $t\theta = u\theta$ an instance of the equation $t = u$.

We give a set of inference rules for deriving consequences of equations. The inference rules are given in the form

$$\frac{A_1, A_2, ..., A_n}{A}$$

indicating that if $A_1, A_2, ..., A_n$ have been derived then we may derive A also using this rule. Later we shall also use such rules in a backward direction, meaning that if we are trying to derive A, we attempt to derive $A_1, A_2, ..., A_n$ in order. We have the following rules:

$$\frac{t = u}{t\theta = u\theta}$$

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

$$\frac{t = u}{f(...t...) = f(...u...)}$$

$$\frac{t = u \qquad u = v}{t = v}$$

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

The following result is due to Birkhoff [Bir35]:

Theorem 6.0.2. *If E is a set of equations then $E \models r = s$ iff $r = s$ is derivable from E using these rules.*

We can state this result in an equivalent way. Namely, $E \models r = s$ iff there is a finite sequence $u_1, u_2, ..., u_n$ of terms such that r is u_1 and s is u_n and for all i, u_{i+1} is obtained from u_i by replacing a subterm t of u_i by a term u, where the equation $t = u$ or the equation $u = t$ is an instance of an equation in E.

This gives us a method for deriving logical consequences of sets of equations. However, it is inefficient. Therefore we are interested in finding restrictions of these inference rules that are still capable of deriving all equational consequences of an equational system. This is the motivation for the theory of term-rewriting systems, which is still a fairly young field.

It is interesting to note that Birkhoff's theorem does not hold for sorted equations, if we allow empty sorts. For a discussion of this see [GM81].

7 Term rewriting

The idea of a term rewriting system is to orient an equation $r = s$ into a rule $r \to s$ indicating that instances of r may be replaced by instances of s but not vice versa. Often this is done in such a way as to replace terms by simpler terms, where the definition of what is simple may be fairly subtle. However, as a first approximation, smaller terms are typically simpler. If we have an equation $x + 0 = x$ then we would typically orient it into a rule $x + 0 \to x$. This reduces the generation of terms like $((x + 0) + 0) + 0$ which can appear in proofs if no such directionality is applied. The study of term rewriting systems is concerned with how to orient rules and what conditions guarantee that the resulting systems have the same computational power as the equational systems they came from. It is clear that some such mechanism is often needed to prevent the generation of unsimplified terms such as that given above. Still, in some work [SP88] we found that use of unrestricted equality axioms as in Birkhoff's theorem was actually more efficient than the use of term rewriting for certain proofs. However, this seems to be an exceptional case.

We expand a little more on the correspondence between proofs and rewriting before getting into more detail. The reason for this is to more fully justify the study of rewriting as a foundational branch of mathematics and not a branch of computation theory. Let's take a very simple example to illustrate the ideas. Suppose the equational system E is $\{a = b, a = c, c = d\}$. Suppose we orient these rules as $b \to a$, $c \to a$, and $d \to c$. Suppose we are trying to show that the equation $b = d$ is a logical consequence of E. We can do this by rewriting $b = d$ using the rule $d \to c$, obtaining $b = c$. Rewriting using the rule $c \to a$ we obtain $b = a$. Finally, rewriting using the rule $b \to a$ we obtain $a = a$. Since we can rewrite both b and d to the same thing, the equation is proved. However, in a formal sense this is unsatisfying, since we have derived $a = a$ from $b = d$ and E. What we wanted to do was to derive $b = d$ from E. If we formalize the inference rules in a slightly different way, this is accomplished.

Corresponding to a rewrite rule $s \to t$ we have the inference rule

$$\frac{A[s\rho] \qquad t \to s}{A[t\rho]}$$

Also, we have $r = r$ as an axiom for all terms r. Here $A[s]$ indicates an equation containing a subterm s. Note that this rule applies the rewrite rule $t \to s$ in a backwards direction, replacing $s\rho$ by $t\rho$ instead of replacing $t\rho$ by $s\rho$. In this system, the above proof of $b = d$ can be expressed this way:

$$\cfrac{\cfrac{\cfrac{a = a \quad b \to a}{b = a} \qquad c \to a}{b = c} \qquad d \to c}{b = d}$$

This proof is not satisfactory either, since it starts from $a = a$ and derives $b = d$. This kind of forward reasoning is often inefficient because it is not sensitive to the structure of the theorem ($b = d$) that we are trying to prove. However, if we use the inference rules in the backwards diretion, this proof makes sense. Our goal is to prove $b = d$. Using the rewrite rule $d \to c$, this is reduced to the subgoal $b = c$. In two more steps, this is reduced to the subgoal $a = a$ which is an axiom, and so the proof is complete. Also, the rewrite rules are used in the proper direction, and the search is sensitive to the theorem we are trying to establish. This way of viewing the rewriting process shows that it can naturally be seen as a restriction on the use of the inference rules in Birkhoff's theorem. This is similar in spirit to Gentzen-style systems [Fit90], which can eliminate connectives from theorems, reducing complex goals to simpler subgoals. The concept of simplicity in Gentzen style systems is largely one of size, where eliminating a connective makes a formula simpler. The concept of simplicity in term rewriting can be much more subtle than this, however. For a discussion of Gentzen-style systems, see chapter **??** of this volume.

8 Terminology

Our notation is slightly simpler than that in [DJ91] in order to make the concepts clearer in this introductory treatment. A *term-rewriting system* R is a set of rules of the form $r \to s$, where r and s are terms. It is common to require that any variable that appears in s must also appear in r. Thus a rule like $f(x) \to g(x, y)$ would not be allowed because the variable y appears on the right but not on the left. Such rules cause problems because y must be instantiated during rewriting, or else rewriting can introduce new variables. However, our formulation permits us to have rewrite rules that do not satisfy this restriction. It is common to require that r is not a variable; our formalism permits this but sometimes we disallow it. The *rewrite relation* $\to_R$ is defined by the following inference rules: We have the following set of inference rules for deriving new statements of the form $s \to t$ for various s and t:

$$\frac{r \to s \qquad \rho \textit{ a substitution}}{r\rho \to s\rho}$$

$$\frac{r \to s}{f(\ldots r \ldots) \to f(\ldots s \ldots)}$$

$$\frac{\textit{true}}{r \to^* r}$$

$$\frac{r \to s}{r \to^* s}$$

$$\frac{r \to^* s \qquad s \to^* t}{r \to^* t}$$

$$\frac{r \to s}{r \leftrightarrow s}$$

$$\frac{s \to r}{r \leftrightarrow s}$$

$$\frac{true}{r \leftrightarrow^* r}$$

$$\frac{r \leftrightarrow s}{r \leftrightarrow^* s}$$

$$\frac{r \leftrightarrow^* s \qquad s \leftrightarrow^* t}{r \leftrightarrow^* t}$$

We use $\vdash_r$ to indicate derivability using these rules. The r subscript refers to "rewriting" (not to the term r). We may think of a set R of rules as a set of logical axioms. If we write $s \to t$ is in R, then we know that $s \to t$ is such an axiom. If we write $R \vdash_r s \to t$ then we know that $s \to t$ may refer to a rewrite relation not included in R. Often we write $s \to_R t$ as an abbreviation for $R \vdash_r s \to t$, and sometimes the subscript R is dropped. We similarly define $\to_R^*$ in terms of derivability from R. Note that the relation $\to_R^*$ is the reflexive transitive closure of $\to_R$. Thus $r \to_R^* s$ if there is a sequence $r_1, r_2, ..., r_n$ such that r_1 is r, r_n is s, and $r_i \to_R r_{i+1}$ for all i. Such a sequence is called a *rewrite sequence* from r to s, or a *derivation* from r to s. Note that $r \to_R^* r$ for all r and R. A term r is *reducible* if there is a term s such that $r \to s$, otherwise r is *irreducible*. If $r \to_R^* s$ and s is irreducible then we call s a *normal form* for r.

For example, suppose we have the system $R = \{x + 0 \to x, 0 + x \to x\}$. Then the term $0 + (y + 0)$ rewrites in two ways; we have $0 + (y + 0) \to 0 + y$ and $0 + (y + 0) \to y + 0$. Applying rewriting again, we obtain $0 + (y + 0) \to^* y$. In this case, y is a normal form of $0 + (y + 0)$, since y cannot be further rewritten. Computationally, rewriting a term s proceeds by finding a subterm t of s such that t is an instance of the left-hand side of some rule in R, and replacing t by the corresponding instance of the right-hand side of the rule. For example, we find that $0 + (y + 0)$ is an instance of the left-hand side $0 + x$ of the rule $0 + x \to x$. The corresponding instance of the right-hand side x of this rule is $y + 0$, so we replace $0 + (y + 0)$ by $y + 0$. This assumes that all variables on the right-hand side appear also on the left-hand side.

We now relate rewriting to equational theories. From the above rules, $r \leftrightarrow s$ if $r \to s$ or $s \to r$, and $\leftrightarrow^*$ is the reflexive transitive closure of $\leftrightarrow$. Thus $r \leftrightarrow^* s$ if there is a sequence $r_1, r_2, ..., r_n$ such that r_1 is r, r_n is s, and $r_i \leftrightarrow r_{i+1}$ for all i. Suppose R is a term rewriting system $\{r_1 \to s_1, ..., r_n \to s_n\}$. We define $R^=$ to be the associated equational system $\{r_1 = s_1, ..., r_n = s_n\}$. Also, we say $t =_R u$ iff $R^= \models t = u$, that is, the equation $t = u$ is a logical consequence of the associated equational system. The relation $=_R$ is thus the smallest congruence relation generated by R, in algebraic terms. The relation $=_R$ is defined semantically, and the relation $\to^*$ is defined syntactically. We would like to find relationships between these two concepts to be able to compute properties of $=_R$ and to find complete restrictions of the inference rules of Birkhoff's theorem. Note that by Birkhoff's theorem, $R^= \models t = u$ iff $t \leftrightarrow_R^* u$. This is already a connection between the two concepts. However, the fact that rewriting can go in both directions in the derivation for $t \leftrightarrow_R^* u$ is a disadvantage. What we will show is that if R has certain properties, some of them decidable, then $t =_R u$ iff any normal form of t is the same as any normal form of u. This permits us to decide if $t =_R u$ by rewriting t and u to any normal form and checking if these are identical.

9 Confluence and termination properties

We now present some properties of term rewriting systems R. Equivalently, these can be thought of as properties of the rewrite relation $\rightarrow_R$. We will adopt both viewpoints at different times, especially when considering rewrite relations based on R other than $\rightarrow_R$. For terms s and t we write $s \downarrow t$ if there is a term u such that $s \rightarrow^* u$ and $t \rightarrow^* u$. We write $s \uparrow t$ if there is a term r such that $r \rightarrow^* s$ and $r \rightarrow^* t$. We say R is *confluent* if for all terms s and t, $s \uparrow t$ implies $s \downarrow t$. The meaning of this is that any two rewrite sequences from a given term, can always be "brought together." Sometimes we are also interested in *ground confluence*. We say R is ground confluent if for all ground terms r, if $r \rightarrow^* s$ and $r \rightarrow^* t$ then $s \downarrow t$. Most research in term rewriting systems concentrates on confluent systems. However, there is also some interest in term rewriting systems that model nondeterminism [Kap88]. For such systems, we may have a nondeterministic choice operator with the rules $choice(x, y) \rightarrow x$ and $choice(x, y) \rightarrow y$. It seems that there is more work to be done in this area. One interesting possibility is to study the denotational semantics of non-confluent systems. See Naoi and Inagaki [NI89] for a study of the denotational semantics of confluent systems.

A term rewriting system R (alternatively, a rewrite relation $\rightarrow$) has the *Church-Rosser property* if for all terms s and t, $s \leftrightarrow^* t$ iff $s \downarrow t$.

Definition 9.0.1. If s and t are two terms then $s \downarrow^1 t$ if $s \downarrow t$ and $s \downarrow^{n+1} t$ if there is a term u such that $s \downarrow u$ and $u \downarrow^n t$.

Lemma 9.0.2. *If R is confluent and $s \downarrow^2 t$ then $s \downarrow t$.*

Proof. Since $s \downarrow^2 t$ there is a term u such that $s \downarrow u$ and $u \downarrow t$. Therefore there are terms v_1 and v_2 such that $s \rightarrow^* v_1$ and $u \rightarrow^* v_1$ and $u \rightarrow^* v_2$ and $t \rightarrow^* v_2$. Since R is confluent and $u \rightarrow^* v_1$ and $u \rightarrow^* v_2$, $v_1 \downarrow v_2$. Therefore there is some term w such that $v_1 \rightarrow^* w$ and $v_2 \rightarrow^* w$. Therefore $s \rightarrow^* v_1 \rightarrow^* w$ and $t \rightarrow^* v_2 \rightarrow^* w$ so $s \downarrow t$. ∎

Theorem 9.0.3. *A term rewriting system R has the Church-Rosser property iff R is confluent.*

Proof. Suppose s and t are terms and $s \leftrightarrow^* t$. We want to show that $s \downarrow t$. Note that if $u \rightarrow^* v$ or $v \rightarrow^* u$ then $u \downarrow v$ (since $v \rightarrow^* v$ and $u \rightarrow^* u$). Hence if $s \leftrightarrow^* t$ then for some n, $s \downarrow^n t$. If s and t are identical or $s \downarrow t$ then we are done. Suppose $s \downarrow^n t$ for $n \geq 2$. We show by induction that $s \downarrow t$. Since $s \downarrow^n t$, there exists a term u such that $s \downarrow^2 u$ and $u \downarrow^{n-2} t$. By lemma 9.0.2, $s \downarrow u$. By the definition of $\downarrow^n$, $s \downarrow^{n-1} t$. By induction, $s \downarrow t$. ∎

Since $s \leftrightarrow^* t$ iff $s =_R t$, this theorem connects the equational theory of R with rewriting. In order to decide if $s =_R t$ it is only necessary to see if s and t have a common normal form.

Definition 9.0.4. A term rewriting system is *terminating* (strongly normalizing) if it has no infinite rewrite sequences. Informally, this means that the rewriting process, applied to a term, will eventually stop, no matter how the rewrite rules are applied.

An example of a terminating system is $\{g(x) \rightarrow f(x), f(x) \rightarrow x\}$. The first rule changes g's to f's and so can only be applied as many times as there are g's. The second rule reduces the size and so it can only be applied as many times as the size of a term. An example of a terminating non-confluent system is the following system for converting a Boolean expression to disjunctive normal form:

$$not(not(x)) \rightarrow x$$

$$not(or(x, y)) \rightarrow and(not(x), not(y))$$

$$not(and(x, y)) \rightarrow or(not(x), not(y))$$

$$and(x, or(y, z)) \rightarrow or(and(x, y), and(x, z))$$

$$and(or(x, y), z) \rightarrow or(and(y, x), and(z, x))$$

The system is not confluent, because the term $and(or(x, y), or(u, v))$ has two distinct normal forms, though they are equivalent up to associativity and commutativity of *and*. A better system would include the associativity and commutativity of *and* and *or* as well as rules for eliminating expressions like $or(x, not(x))$. This cannot be done without modifying the definition of rewriting, since commutativity leads to nontermination; for a discussion of this see section 15.1. Even allowing for associativity and commutativity of *and* and *or*, Socher shows in [SA91] that there is no equivalent terminating and confluent system. We say two term rewriting systems are *equivalent* if their associated equational theories are equivalent (have the same logical consequences). An example of another terminating system for Boolean algebra is the following, which uses exclusive *or* (*xor*) as another connective:

$$and(x, true) \rightarrow x$$

$$and(x, false) \rightarrow false$$

$$and(x, x) \rightarrow x$$

$$xor(x, false) \rightarrow x$$

$$xor(x, x) \rightarrow false$$

$$and(xor(x, y), z) \rightarrow xor(and(x, z), and(y, z))$$

We also have the associativity and commutativity of *and* and *xor*:

$$and(x, y) = and(y, x)$$

$$and(x, and(y, z)) = and(and(x, y), z)$$

$$xor(x, y) = xor(y, x)$$

$$xor(x, xor(y, z)) = xor(xor(x, y), z)$$

For this system, we have to modify the operation of rewriting as indicated above, to permit the associativity and commutativity of *and* and *xor* to be used before every rewriting step. This system is nearly confluent; it will reduce two expressions to the same form up to associativity and commutativity of *and* and *xor* iff they are equivalent. For example, to show that $xor(and(a, b), and(b, a)) = false$ we need to use the commutativity of *and* and apply the rule $xor(x, x) \to false$.

An example of a nonterminating system is $\{x \to f(x)\}$. It can be difficult to determine if a system is terminating. The intuitive idea is that a system terminates if each rule makes a term simpler in some sense. However, the definition of simplicity is not always related to size. It can be that a term becomes simpler even if it becomes larger. In fact, it is not even partially decidable whether a term rewriting system is terminating [HL78]. Termination orderings are often used to prove that term rewriting systems are terminating.

Definition 9.0.5. A partial ordering $>$ is *well-founded* if there are no infinite sequences x_i of elements such that $x_i > x_{i+1}$ for all $i \geq 0$.

Definition 9.0.6. A *termination ordering* on terms is a partial ordering $>$ which is well founded and satisfies the *full invariance property*, that is, if $s > t$ and Θ is a substitution then $s\Theta > t\Theta$, and also satisfies the *replacement property*, that is, $s > t$ implies $f(...s...) > f(...t...)$.

Note that if $s > t$ and $>$ is a termination ordering then all variables in t appear also in s. For example, if $f(x) > g(x, y)$, then by full invariance $f(x) > g(x, f(x))$, and by replacement $g(x, f(x)) > g(x, g(x, f(x)))$, et cetera, giving an infinite descending sequence of terms.

Theorem 9.0.7. *Suppose R is a term rewriting system and $>$ is a termination ordering and for all rules $r \to s$ in R, $r > s$. Then R is terminating.*

Proof. From the hypotheses it follows that if $t \to_R u$ then $t > u$. Since $>$ is well founded, there can be no infinite R rewrite sequences. ∎

This result can be extended to quasi-orderings, which are relations that are reflexive and transitive (see Section 14), but the above result should be enough to give an idea of the proof methods used. Many termination orderings are known; some will be discussed in Section 14. The orderings of interest are computable orderings, that is, it is decidable whether $r > s$ given terms r and s.

Note that if R is terminating, we can always find a normal form of a term by any rewrite sequence continued long enough. However there can be more than one normal form. If R is terminating and confluent, there is only one normal form and always one normal form for every term. This gives a decision procedure for the equational theory, since for terms r and s, $r =_R s$ iff $r \leftrightarrow_R^* s$ (by Birkhoff's theorem) iff $r \downarrow s$ (by confluence) iff r and s have the same normal form (by termination). This gives us a directed form of theorem proving in such an equational theory. A term rewriting system which is both terminating and confluent is called *canonical*. Some authors use the term *convergent* for such systems [DJ90]. Many such systems are known. Here are some examples. First we give a two rule system for the *append* function on lists:

$$append(cons(x, y), z) \rightarrow cons(x, append(y, z))$$

$$append(nil, z) \rightarrow z$$

Similar canonical systems could be given for many other functions on lists. Now we give a simple system for multiplying out sums and eliminating zeroes and ones when possible:

$$times(x, plus(y, z)) \rightarrow plus(times(x, y), times(x, z))$$

$$times(plus(x, y), z) \rightarrow plus(times(x, z), times(y, z))$$

$$times(0, x) \rightarrow 0$$

$$times(x, 0) \rightarrow 0$$

$$times(1, x) \rightarrow x$$

$$times(x, 1) \rightarrow x$$

$$plus(x, 0) \rightarrow x$$

$$plus(0, x) \rightarrow x$$

$$plus(plus(x, y), z) \rightarrow plus(x, plus(y, z))$$

In the following discussion, we abbreviate $plus(x, y)$ and $times(x, y)$ as $x + y$ and $x * y$, respectively, as usual. This system is confluent up to the associativity and commutativity of addition. However, it is not confluent because the term $(x + y) * (u + v)$ can be rewritten by applying either distributivity law first, yielding eventually the results $x*u+y*u+x*v+y*v$ and $x*u+x*v+y*u+y*v$, which are irreducible and not identical. If we add the rules $x+y \rightarrow y+x$ and $x+(y+z) \rightarrow y+(x+z)$ then the system is confluent, since the order of elements in a sum can be changed. However, the system is no longer terminating. It is still *globally finite*, which means that for every term s there are finitely many terms t such that $s \rightarrow^* t$. For a discussion of global finiteness, see [GKM83]. It can be shown that all terminating finite term rewriting systems are globally finite. Finally, we give some recursive definitions for plus and times that are canonical, where s(x) means x+1:

$$plus(0,x) \rightarrow x$$

$$plus(s(x),y) \rightarrow s(plus(x,y))$$

$$times(0,x) \rightarrow 0$$

$$times(s(x),y) \rightarrow plus(y, times(x,y))$$

In a similar way, we could present canonical term rewriting systems for any primitive recursive function.

The concept of a *multiset* is often useful to show termination. Informally, a multiset is a set in which an element can occur more than once. Formally, a multiset S is a function from some underlying domain D to the non-negative integers. It is said to be finite if $\{x : S(x) > 0\}$ is finite. We say $x \in S$ if $S(x) > 0$. We call $S(x)$ the *multiplicity* of x in S; this represents the number of times x appears in S. If S and T are multisets then $S \cup T$ is defined by $(S \cup T)(x) = S(x) + T(x)$ for all x. A partial ordering $>$ on D can be extended to a partial ordering $\gg$ on multisets in the following way: We say $S \gg T$ if there is some multiset V such that $S = S' \cup V$ and $T = T' \cup V$ and for all t in T' there is an s in S' such that $s > t$. This relation can be computed reasonably fast by deleting common elements from S and T as long as possible, then testing if the specified relation between S' and T' holds. The idea is that a multiset becomes smaller if an element is replaced by any number of smaller elements. Thus $\{3,4,4\} \gg \{2,2,2,2,1,4,4\}$ since 3 has been replaced by $2,2,2,2,1$. This operation can be repeated any number of times, still yielding a smaller multiset; in fact, the relation $\gg$ can be defined in this way as the smallest transitive relation having this property [Der87]. One can show that if $>$ is well founded, so is $\gg$. For a comparison with other definitions of multiset ordering, see [JL82].

We have indicated how termination is shown; more will be presented in Section 14. However, we haven't shown how to prove confluence. As stated, this looks like a difficult property. However, it turns out that if R is terminating, confluence is decidable, from Newman's lemma [New42], given below. If R is not terminating, there are some methods that can still be used to prove confluence. This is interesting, even though in that case we do not get a decision procedure by rewriting to normal form, since it allows some flexibility in the rewriting procedure.

Definition 9.0.8. A term rewriting system is *locally confluent (weakly confluent)* if for all terms r, s, and t, if $r \rightarrow s$ and $r \rightarrow t$ then $s \downarrow t$.

Theorem 9.0.9 (Newman's lemma). *If R is locally confluent and terminating then R is confluent.*

Proof. We use the idea of proof orderings and multiset orderings, similar to the methods used in [BDH86]. However, the ordering we use is simpler, and is from [Der84, KMN88]. We show a stronger property than needed; namely, we show that if $s \leftrightarrow^* t$ and R is locally confluent and terminating then $s \downarrow t$. Suppose $s \leftrightarrow^* t$. Then there is some sequence $r_1, r_2, ..., r_n$ of terms such that s is r_1 and t is r_n and for all i, $r_i \leftrightarrow r_{i+1}$. For such a proof, consider the multiset S in which $S(u)$ is the number of times the term u appears among the r_i. Let the terms be ordered by $>$, defined by $u > v$ if $u \rightarrow^+ v$, that is, v is obtained from u by one or more rewrites. Since R is terminating, this ordering is well-founded. Therefore the ordering $\gg$ on multisets of terms is also well-founded.

If for no term r_i do we have $r_{i-1} \leftarrow r_i \rightarrow r_{i+1}$ then $s \downarrow t$ immediately. Suppose for some r_i we have $r_{i-1} \leftarrow r_i \rightarrow r_{i+1}$. Since R is locally confluent, $r_{i-1} \downarrow r_{i+1}$. Thus there is another derivation between s and t in which r_i is replaced by terms in the derivation $r_{i-1} \downarrow r_{i+1}$. These terms are all smaller than r_i in the ordering $>$. Let T be the multiset for this new derivation. Since r_i in S has been replaced by smaller terms, $S \gg T$. By induction on the ordering $\gg$, we can assume that the derivation for T can be converted to the desired form, that is, $s \rightarrow s_1 \rightarrow s_2 \rightarrow ... \leftarrow t_2 \leftarrow t_1 \leftarrow t$, so $s \downarrow t$. This completes the proof. ∎

Next we discuss methods of showing that R is locally confluent, using "critical pairs." Also, we discuss methods of adding rules to a non-confluent system to attempt to create an equivalent confluent system. In addition, we discuss methods of showing confluence for systems that are non-terminating; such systems often arise in modeling programs by term-rewriting systems, since termination is often not known for programs.

10 Tests for confluence

We still have the problem of showing that R is locally confluent. This can be done using *critical pairs*. To show R is locally confluent, we need to know that for all terms t, if t rewrites in two different ways, then these two different results have a common normal form. We first consider the relationship between different rewrites of t. We call a subterm of t that is an instance of the left-hand side of a rule in R, a *redex*. Each way of rewriting a term t corresponds to a redex of t, although the same redex may be rewritten in more than one way. Suppose a term rewrites in two different ways. One possibility is that the redexes for these rewrites are disjoint subterms of t. Suppose t is $f(a, b)$ and R contains the rules $\{a \rightarrow c, b \rightarrow d\}$, among others. Then we can rewrite either the redex a or b, obtaining $f(c, b)$ and $f(a, d)$, respectively. In this case, we can do the other rewrite next. For example, on $f(c, b)$ we can use the rewrite $b \rightarrow d$ to obtain $f(c, d)$, and similarly $f(a, d)$ rewrites to $f(c, d)$. The second possibility is that one redex is a subterm of the other. For example, t may be $g(a)$ and R may include $\{a \rightarrow b, g(x) \rightarrow f(x, x)\}$. Then t rewrites to $g(b)$ using the first rule on the redex a and to $f(a, a)$ using the second rule on the redex $g(a)$. Now, $g(b)$ rewrites to $f(b, b)$ using the second rule, and $f(a, a)$ rewrites to $f(b, b)$ using two applications of the first rule. Note in this case that the two redexes of t are a and $g(a)$, one being a subterm of the other. In addition, the redex a corresponds to the position of x in the rule $g(x) \rightarrow f(x, x)$. In general, the second possibility occurs when one redex is a subterm of the other, and corresponds to a variable position in the other rule, or a subterm of such a variable position. (We will explain this more formally below.) The final possibility is when the two rules "overlap." This means that one redex is a subterm of the other, but does not correspond to a variable position of the other rule. For example, we may have the term t as $(x * e) * z$ and R as $\{(x * y) * z \rightarrow x * (y * z), x * e \rightarrow x\}$. (We are using infix notation $x * y$ instead of prefix notation $*(x, y)$ for the operator $*$.) Here the two redexes are $(x * e) * z$ and $x * e$. From the two rules we obtain the terms $x * (e * z)$ and $x * z$, respectively. No more rewriting can be done on either term.

In general, these are the only possibilities for a term to rewrite in two different ways. The first two ways always produce a pair of terms with a common normal form. The only kind of a choice that can violate local confluence is the third possibility, when one redex is a subterm of another but in a non-variable position of the other rule. This is called an *overlap*, and the pair of terms resulting from this is called a *critical pair*. So, we have the result that R is locally confluent if all its critical pairs have a common normal form. We now state this more precisely.

We introduce some formalism for specifying occurrences of subterms of a term. If s is a term, then let $\overline{\overline{s}}$, the *string* of s, be the term as a sequence $\alpha_1\alpha_2...\alpha_n$ of characters, omitting commas and parentheses. Thus $\overline{\overline{f(x, g(y))}}$ is $fxgy$, a string of four characters. If s has varyadic symbols, we may have to include parentheses in $\overline{\overline{s}}$ to distinguish them. If α is $\overline{\overline{s}}$ for some s then s is unique, and we define $\underline{\underline{\alpha}}$ to be s. Thus $\underline{\underline{fxgy}}$ is $f(x, g(y))$, and this is an inverse of the string operator on terms. A *subterm occurrence* of s is a substring of $\overline{\overline{s}}$ that is a term. If $\alpha\overline{\overline{t}}\gamma$ is $\overline{\overline{s}}$ then we specify this term and its subterm occurrence t as the triple $(\alpha, \overline{\overline{t}}, \gamma)$. We call (α, γ) the *context* of the subterm occurrence. Thus, the triple (f, x, gy) specifies the occurrence of x in the term $f(x, g(y))$. Often the context is ignored, so we can say that one subterm occurrence is a subterm of another, etc. Sometimes we call a term $r = \underline{\underline{\alpha\Box\gamma}}$ a context. If r is a context $\underline{\underline{\alpha\Box\gamma}}$ and u is a term then $r[u]$ denotes the term $\underline{\underline{\alpha\overline{\overline{u}}\gamma}}$, that is, r with the occurrence of $\Box$ replaced by u. Sometimes we identify a subterm u with $\overline{\overline{u}}$. It is easy to show that if α and β are two subterm occurrences, then either they are disjoint or one is a subterm (substring) of the other. Finally, suppose $\overline{\overline{t}}$ is $\gamma_1 x_1 \gamma_2 x_2 ... \gamma_n x_n \gamma_{n+1}$ where the x_i are (not necessarily distinct) variable occurrences and the γ_i are substrings that do not contain variables. Suppose Θ is a substitution that replaces x_i by term t_i. Thus $\overline{\overline{t\Theta}}$ is $\gamma_1\overline{\overline{t_1}}\gamma_2\overline{\overline{t_2}}...\gamma_n\overline{\overline{t_n}}\gamma_{n+1}$. Then we call the subterm occurrence t_i corresponding to x_i in $\overline{\overline{t\Theta}}$ a *variable instance occurrence* for (t, Θ). For example, if t is $f(x, g(y))$ and $t\Theta$ is $f(a, g(g(b)))$ then the subterm occurrences a and $g(b)$ are variable instance occurrences for (t, Θ). This notation is, we feel, more intuitive than the usual "string of integers" notation for identifying subterm occurrences. For a similar idea, see [KB70].

We now indicate the three possible ways in which redexes can overlap. Suppose t has two redexes $l_1\Theta$ and $l_2\Theta$, instances of terms l_1 and l_2, respectively. Suppose $\overline{\overline{l_1}}$ is $\gamma_1 x_1 \gamma_2 x_2 ... \gamma_n x_n \gamma_{n+1}$ where the x_i are (not necessarily distinct) variables and none of the γ_i contain variables. Suppose $l_1\Theta$ is the larger of these redexes (or they have the same size), and $\overline{\overline{l_1\Theta}}$ is $\gamma_1 t_1 \gamma_2 t_2 ... \gamma_n t_n \gamma_{n+1}$. Then $\overline{\overline{t}}$ is $\alpha\gamma_1 t_1 \gamma_2 t_2 ... \gamma_n t_n \gamma_{n+1}\beta$ for some α and β. There are three places in which the redex $l_2\Theta$ can occur: 1) It can be a substring of α or β; this means that the redexes are disjoint. 2) It can be a substring of some t_i. 3) It can be a substring of the redex $l_1\Theta$ but not a substring of t_i for any i; this corresponds to a critical pair. Note that $l_2\Theta$ cannot properly contain $l_1\Theta$ since $l_1\Theta$ is larger. There are no other possibilities, since as noted above, two subterm occurrences are either disjoint or one is a subterm of the other.

Critical pairs are based on most general unifiers. We say that a substitution α is *as general as* a substitution β relative to a pair $(s,\ t)$ of terms if there is a substitution γ such that $s\alpha\gamma$ is identical to $s\beta$ and $t\alpha\gamma$ is identical to $t\beta$. A substitution α is a *unifier* of two terms s and t if $s\alpha$ is identical to $t\alpha$. In this case we say that s and t are *unifiable*. A substitution α is a *most general unifier* of terms s and t if α is a unifier of s and t and α is as general as any other unifier of s and t, relative to the pair $(s,\ t)$. It turns out that if s and t are unifiable then such most general unifiers exist and are easy to compute [Rob65, JK91].

In the following definition, and in general, we often use "hats" to indicate instances. Thus $\hat{x}$ is an instance of x, $\hat{u}$ is an instance of u, et cetera. The substitution applied should be apparent from context.

Definition 10.0.1. Suppose R is a term rewriting system and $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ are two rules in R without common variables. (If necessary, the variables can be renamed to insure this.) Suppose u is a subterm of l_1 and u is not a variable (unless l_1 is a variable, then u can be l_1). Suppose u and l_2 are unifiable, and let Θ be a most general unifier. Suppose $\overline{\overline{l_1}}$ is $\gamma_1 u \gamma_2$. Assume that either the rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ are distinct or u is different from l_1 (this prohibits trivial critical pairs of a rule with itself). We call the pair $(\underline{\underline{\hat{\gamma_1}\hat{r_2}\hat{\gamma_2}, \hat{r_1}}})$ a *critical pair* for the rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$, or, for R. We call u the *critical subterm occurrence* of l_1. We call Θ the *critical substitution*.

The idea of this is that $\overline{\overline{l_1\Theta}}$ is $\hat{\gamma_1}\hat{u}\hat{\gamma_2}$. Note that $\hat{u}$ is $u\Theta$, which is $l_2\Theta$ so the rewrite rule instance $l_2\Theta \rightarrow r_2\Theta$ may be applied to the redex $u\Theta$, yielding $\hat{\gamma_1}\hat{r_2}\hat{\gamma_2}$. Also, the rule $l_1\Theta \rightarrow r_1\Theta$ can be applied to the term $l_1\Theta$ yielding the term $r_1\Theta$. Thus the critical pair represents two different ways in which the term $l_1\Theta$ can be rewritten.

For example, consider the two rules $g(u) * u \rightarrow 1$ and $(x * y) * z \rightarrow x * (y * z)$. We can unify $g(u) * u$ with a subterm of $(x * y) * z$, obtaining the term $(g(u) * u) * z$. This term rewrites to $1 * z$ using the rule $g(u) * u \rightarrow 1$ and to $g(u) * (u * z)$ using the rule $(x * y) * z \rightarrow x * (y * z)$. This gives us the critical pair $(g(u) * (u * z), 1 * z)$. As another example, the two rewrite rules $f(f(u,v), g(v,u)) \rightarrow h(u,v)$ and $g(f(x,y), g(y,x)) \rightarrow k(x,y)$ have four critical pairs, obtained by unifying the left-hand sides with a subterm of themselves or with a subterm of the other left-hand side. One of the critical pairs is $(g(h(u,v), g(g(v,u), f(u,v))), k(f(u,v), g(v,u)))$.

Theorem 10.0.2 (Critical pair theorem). *Suppose a term rewriting system R is terminating. Then R is confluent iff for all critical pairs (s,t) of R, for arbitrarily chosen normal forms s' and t' of s and t, respectively, s' and t' are identical. Note that we don't require that all normal forms be identical, just that some arbitrarily chosen normal forms be identical.*

Proof. Suppose R is confluent. Then every term has exactly one normal form. Thus s and t have one normal form. Since $s \uparrow t$, $s \downarrow t$. Thus arbitrarily chosen normal forms of s and t will be identical.

Suppose that for all critical pairs (s,t), the stated condition on normal forms holds. We want to show that R is locally confluent. From this and Newman's lemma [New42], it follows that R is confluent. To show that R is locally confluent, we need to consider all possible ways that a term r can rewrite into two terms r_1 and r_2 in one step. There are three possibilities, as indicated above: 1) The redexes for $r \to r_1$ and $r \to r_2$ are disjoint. 2) The redex for $r \to r_2$ is a subterm of a variable occurrence instance for $r \to r_1$ (or vice versa). 3) The redex for $r \to r_2$ is a subterm of the redex for $r \to r_1$ (or vice versa), but not a variable occurrence instance. As argued above, these are the only possibilities, and for cases 1) and 2), we know that r_1 and r_2 have a common normal form. For case 1), since the redexes are disjoint, the rewrite rules can be applied in either order, with the same result. For example, if $a \to b$ and $c \to d$ then the term $f(a,c)$ rewrites to both $f(b,c)$ and $f(a,d)$. The redexes a and c are disjoint. It only takes one more rewrite to obtain $f(b,d)$ from either $f(b,c)$ or $f(a,d)$. Thus $r_1 \downarrow r_2$. For case 2), r_1 and r_2 rewrite to a common term, but it may require more than one rewrite if a variable occurs more than once in a right-hand side of a rewrite rule. For example, suppose $f(x) \to g(x,x)$ and $a \to b$ are in R. Then the term $f(a)$ rewrites to both $g(a,a)$ and $f(b)$. These terms both rewrite to $g(b,b)$, but it takes two rewrites to rewrite $g(a,a)$ to $g(b,b)$ since two occurrences of a must be rewritten. Only one rewrite of $f(b)$ is needed to obtain $g(b,b)$. So, $r_1 \downarrow r_2$ in case 2). It turns out that in case 3), there must be a context (α,β) and terms s, s_1, and s_2 such that $\overline{\overline{r}}$ is $\alpha\overline{\overline{s}}\beta$ and $\overline{\overline{r_1}}$ is $\alpha\overline{\overline{s_1}}\beta$ and $\overline{\overline{r_2}}$ is $\alpha\overline{\overline{s_2}}\beta$ and $s \to s_1$ and $s \to s_2$, that is, $s \to s_1$ is an instance of a rule in R and s_2 is obtained by rewriting a subterm of s in one step using R. In this case, (s_1, s_2) must be an instance of a critical pair for R. That is, there is a critical pair (t_1, t_2) and a substitution Θ such that (s_1, s_2) is $(t_1\Theta, t_2\Theta)$. We need to show that r_1 and r_2 have a common normal form. We have assumed that t_1 and t_2 have a common normal form, hence $t_1\Theta$ and $t_2\Theta$ do, hence s_1 and s_2 do. Thus in all cases, we find a common normal form for r_1 and r_2, so R is locally confluent. ∎

The implication of this result is that if R is a (finite) term rewriting system and R is terminating, then the local confluence (and confluence) of R is decidable, since for a finite R, there are only finitely many critical pairs, and they can be computed (in polynomial time), given R. However, often we are interested in systems that are not necessarily known to terminate, such as term rewriting systems for functional programs. For such systems, other methods can be applied to prove confluence.

It is interesting that confluence of ground systems is decidable [DTHL87]. However, ground confluence (defined in Section 9) is not decidable, as shown by Kapur et al [KNO90]. This property is useful for the so-called "inductionless induction" method [Mus80, HH80, Gog80].

10.1 Orthogonal systems

Definition 10.1.1. A term is *linear* if it has no repeated variables. Thus $f(x,y)$ is linear but $f(x,x)$ is not. A term rewriting system R is *left-linear* if for all rules $l \to r$ in R, l is linear. Such a rule $l \to r$ is also called left-linear.

Definition 10.1.2. A term rewriting system R is *non-overlapping* if it has no critical pairs. R is *orthogonal* (regular) if R is non-overlapping and all rules in R are left-linear. We also require that all variables in the right-hand sides of rules appear also in the left-hand sides, for orthogonal systems.

Note that no critical pairs means that among the left-hand sides, no left-hand side unifies with a subterm of any other left-hand side, or with a proper subterm of itself. The left-linearity restriction is reasonable, since the formal parameters of procedures are distinct. It is also convenient for pattern matching; to check if a rule is an instance of $f(x, x)$ it is necessary to see if it is of the form $f(s, t)$ for terms s and t, and then check that s and t are identical. This test can be expensive, since it may require time proportional to the size of s or t. However, there are also cases where it is convenient to use non-left-linear rules.

Here is a simple example of an orthogonal system, for computing addition and multiplication of integers in successor notation, and for computing bounded subtraction.

$$plus(0, X) \rightarrow X$$

$$plus(s(X), Y) \rightarrow s(plus(X, Y))$$

$$times(0, X) \rightarrow 0$$

$$times(s(X), Y) \rightarrow plus(Y, times(X, Y))$$

$$minus(0, X) \rightarrow 0$$

$$minus(s(X), 0) \rightarrow s(X)$$

$$minus(s(X), s(Y)) \rightarrow minus(X, Y)$$

Note that if the equation $minus(s(X), 0) \rightarrow s(X)$ had been replaced by $minus(X, 0) \rightarrow X$ then the system would be overlapping, since $minus(X, 0)$ and $minus(0, X)$ unify. There is a slight generalization of the concept of non-overlapping that allows this kind of a unification.

Theorem 10.1.3. *If R is orthogonal then R is confluent.*

Proof. The idea is to show that a parallel rewriting relation associated with R is "strongly confluent," defined below. Parallel rewriting is rewriting at one or more disjoint redexes at the same time. We need to consider parallel rewriting, because if s rewrites to t_1 and s rewrites to t_2, then a subterm of s may appear many times in t_1 or t_2, and all of these occurrences may have to be rewritten in parallel to find a t to which both t_1 and t_2 rewrite in one (parallel) step. For example, if s is $f(a)$ and R is $\{a \rightarrow b, f(x) \rightarrow g(f(x), f(x))\}$ then $f(a)$ rewrites to both $f(b)$ and $g(f(a), f(a))$. Both of these terms parallel rewrite to $g(f(b), f(b))$, and this requires parallel rewriting of the two occurrences of a in $g(f(a), f(a))$. Note that we do not assume that R terminates. Similar techniques are used to show the confluence of lambda calculus [Bar84] and combinatory logic [CF58], which do not terminate. ∎

Definition 10.1.4. A term rewriting system R is *strongly confluent* if for all terms r, s, and t, if $r \rightarrow s$ and $r \rightarrow t$ then either s and t are identical or $s \rightarrow t$ or $t \rightarrow s$ or there is a term u such that $s \rightarrow u$ and $t \rightarrow u$.

Theorem 10.1.5. *If R is strongly confluent then R is confluent [New42].*

Proof. The proof is by induction on length of rewrite sequences. We invite the reader to construct the proof, which is straightforward. ∎

Actually, the proof that orthogonal systems are confluent uses the strong confluence of a particular rewrite relation associated with R, not the strong confluence of R itself. In general, the properties of confluence, termination, etc. can be associated with an arbitrary binary relation, not just with term rewriting. For example, we can say that a binary relation R is terminating if there are no infinite sequences x_i such that for all i, $R(x_i, x_{i+1})$. This viewpoint is adopted in the survey by Huet [Hue80] and elsewhere. Then, we can talk about parallel rewriting being confluent, because it is meaningful to define the confluence of the relation $R(x,y)$ defined as "x parallel rewrites to y in one step."

The following well-known example shows that the left-linearity requirement is essential. This example is from [Klo80]. For other examples, see Dershowitz and Jouannaud [DJ90].

$$d(x,x) \to e$$

$$c(x) \to d(x, c(x))$$

$$a \to c(a)$$

The system is non-overlapping because the outermost function symbols of each left-hand side are distinct, and there are no non-variable proper subterms. Note that $c(a) \to^* e$ and $c(a) \to^* c(e)$ but we do not have $e \downarrow c(e)$.

Even though non-left-linearity removes the confluence guarantee, Chew [Che81] showed that some limited confluence properties still apply to some non-overlapping non-left-linear systems, in terms of the uniqueness of normal forms. For example, his methods show that the above system has unique normal forms, despite its lack of confluence.

11 Completion

If a term rewriting system is not confluent, it may be possible to add rules to it to obtain an equivalent confluent system. Recall that two systems are equivalent if they have the same logical consequences, considered as sets of equations. Thus we can change the particular term rewriting system to make it more useful, while still considering the same underlying equational theory. This transformation is not often done for nonterminating left-linear systems; it is more often done for terminating systems. This is done by adding rules while preserving termination and gaining confluence. The idea is that if (s,t) is a critical pair of R, then $R^= \models s = t$, so the rule $s \to t$ or $t \to s$ can be added without changing the equational theory. This rule will make the pair (s,t) rewrite to a common term. If enough such rules can be added while preserving termination, we may end up with a canonical term rewriting system. This idea was proposed in Knuth-Bendix [KB70], and a similar idea is from [Buc87] in the context of polynomial algebra. Of course, each new rule added may cause more critical pairs to be generated, so the process may not terminate. Another possibility is that we cannot orient the rule either way and preserve termination; in this case we say that the completion procedure *fails*. Thus this procedure may either *succeed*, *loop* (fail to terminate), or *fail*. Also, we may want to simplify (rewrite) the rules of R as we go. Even if completion loops, it can sometimes be useful for theorem proving. Some *unfailing completion* methods have been developed that are guaranteed not to fail. There has also been some work in finding restrictions on the generation of critical pairs that do not reduce the effectiveness of completion.

We give some very simple examples to illustrate the idea. Suppose we have the rules

$$c \to b$$

$$f \to b$$

$$f \to a$$

$$e \to a$$

$$e \to d$$

This system is not confluent since $f \to b$ and $f \to a$ but b and a do not rewrite to a common term. From this pair of rules we obtain the critical pair (b, a). From $e \to a$ and $e \to d$ we obtain the critical pair (a, d). Suppose we use the alphabetic ordering, so that $b > a$ since a occurs earlier in the alphabet than b. We orient the critical pairs using this ordering and add the rules $b \to a$ and $d \to a$. Now the set of rules is

$$c \to b$$

$$f \to b$$

$$f \to a$$

$$e \to a$$

$$e \to d$$

$$b \to a$$

$$d \to a$$

Now everything rewrites to a, and the system is confluent and terminating. In general we assume that there is a well-founded ordering $>$ such that if $R \vdash_r s \to t$ then $s > t$; this insures that $\to$ is terminating. We orient a critical pair (s, t) as $s \to t$ if $s > t$ and as $t \to s$ if $s < t$. Otherwise, this rule cannot be oriented, and the completion fails. In practice, the ordering $>$ is a partial ordering that is not fully known initially. As completion proceeds, this ordering may be strengthened, to permit the orientation of the critical pairs that arise. This is the so-called "incremental property" of an ordering [JLR82].

To understand the working of this Knuth-Bendix completion method, we consider its effect on a rewrite proof between c and d. Note that if $R^= \models s = t$ then by Birkhoff's theorem there is a rewrite proof between s and t, that is, a proof using forward and backward rewriting. For example, since $R^= \models c = d$, there is a proof between c and d, and the following is such a proof:

$$c \rightarrow b \leftarrow f \rightarrow a \leftarrow e \rightarrow d$$

When we add the rules $b \rightarrow a$ and $d \rightarrow a$ this proof is shortened to

$$c \rightarrow b \rightarrow a \leftarrow d$$

Note that the critical pair (b, a) covers the part of the sequence $b \leftarrow f \rightarrow a$ and when the rule $b \rightarrow a$ is added, this subsequence is replaced by $b \rightarrow a$. The final proof is a "valley," that is, of the form $\rightarrow\rightarrow\rightarrow \ldots \leftarrow\leftarrow\leftarrow$. This terminology was used, for example, in [KMN88]. The Knuth-Bendix procedure has the effect of changing arbitrary rewrite sequences to valleys. This is accomplished by eliminating sub-proofs of the form $\leftarrow\rightarrow$ from rewrite proofs. The idea of the proof ordering work of [BDH86] is to define a complexity ordering on proofs such that each completion step makes the proof simpler. A first approximation to the complexity ordering is just to consider the proof as a multiset of terms, ordered by the extension of the ordering $>$ to multisets. However, it turns out that this is not sufficient to deal with the rewriting of rewrite rules using other rewrite rules (not explained yet), so a more complicated ordering is used. This ordering considers not only the sequence of terms but also which equation or rewrite rule is used to obtain each term from an adjacent term in the sequence, and at which subterm the equation or rule is applied, and what substitution is used. For details, see the references [BDH86] and [BDP89]. The idea is that the complexity ordering is well-founded, so there is no infinite sequence of proofs, each simpler than the previous one. Also, given any proof from R, there is still a proof from R', where R' results from R using one inference step, and the proof from R' is as simple or simpler than the proof from R. In addition, there is some inference on R that yields an R' making the proof simpler. Therefore, some kind of fairness assumption is needed to insure that each proof will eventually be simplified as much as possible. Fairness means that any inference that is henceforth possible will eventually be performed. More precisely, no inference is henceforth possible and never performed. A proof that cannot be simplified is said to be in *normal form*. For the Knuth-Bendix procedure, a proof is in normal form if it is a valley. In general, we can show that any fair completion method for which such an ordering exists, will eventually produce a proof in normal form. These proofs are particularly simple, and can be detected easily; when such a proof is found, a completion program can halt. For example, when a valley proof of an equation $s = t$ is found, then s and t rewrite to the same normal form, and this can easily be detected. The Knuth-Bendix procedure eliminates sub-proofs of the form $\leftarrow\rightarrow$, so we know that a normal form proof contains no such sub-proofs, and therefore must be a valley. Usually, completion procedures can be used as theorem provers, even when they fail to halt. For example, if a valley proof of $s = t$ is found, then we can detect this and halt, even though the completion procedure may still have more inferences to apply. In this way, completion procedures can be used as theorem provers even when they generate an infinite set of rewrite rules.

We introduce some notation for proofs to simplify the discussion. For a rewrite proof, we define a *proof trace* to be the sequence of rewrite relations in the proof. Thus the proof $c \rightarrow b \leftarrow f \rightarrow a \leftarrow e \rightarrow d$ has the trace $\rightarrow\leftarrow\rightarrow\leftarrow\rightarrow$. In general, a proof trace will be a sequence of $\{\rightarrow, \leftarrow, =\}$, where the use of "=" will be explained below. A *proof pattern* is a set of proof traces. Often we identify a proof trace with a proof pattern containing only that proof trace. In a proof pattern, we use an asterisk to indicate zero or more occurrences, as in regular expressions in automata theory. Thus $\rightarrow^*$ indicates zero or more occurrences of $\rightarrow$. In this notation, a *valley* is a proof whose trace is in the pattern $\rightarrow^* \leftarrow^*$.

The example of completion given above is misleading in that completion procedures typically simplify the rewrite rules. Thus the rule $f \rightarrow a$ could be used to rewrite the left-hand side of the rule $f \rightarrow b$, giving the rule $a \rightarrow b$, which could be re-oriented as $b \rightarrow a$, using the alphabetic ordering mentioned above. Similarly, the rule $d \rightarrow a$ could be generated by simplifying the rule $e \rightarrow a$ using the rule $e \rightarrow d$. However, even without such simplification of rules, this example does illustrate the working of the completion method.

Sometimes a proof can be transformed into a valley by use of the critical pair theorem, without any completion steps at all. For example, if $R = \{b \rightarrow a, d \rightarrow c\}$ then the following is a proof between $f(b, c)$ and $f(a, d)$:

$$f(b, c) \leftarrow f(b, d) \rightarrow f(a, d)$$

The two rewrites can be exchanged as in the critical pair theorem, giving the proof

$$f(b, c) \rightarrow f(a, c) \leftarrow f(a, d)$$

In general, a proof can be converted to a valley proof by a sequence of such exchanges and completion steps.

Theorem 11.0.1. *If fair completion starting from set R of rewrite rules and using the same ordering throughout the completion (or one of increasing strength) does not fail, and if $R \models s = t$, then it will eventually produce a valley proof of $s = t$. Furthermore, let S be the set of rules that are generated and never henceforth deleted during completion. Then S is confluent and equivalent to R.*

The termination ordering may become stronger, in order to orient critical pairs (u, v) that cannot be ordered using the current ordering. Usually the ordering is chosen as weak as possible at the beginning, and strengthened as needed during completion. Note that since there is a valley proof of $s = t$, this means that s and t rewrite to a common term. Since S is confluent, this means that s and t have unique S-normal forms, which are identical. Thus, s and t can be rewritten in any way during the completion process, and they will eventually become identical, as long as they are kept in normal form relative to the current set of rules.

11.1 Completion with equations

Knuth-Bendix completion can be extended to the so-called *unfailing completion* first introduced in [BDP89]. For this we have to let R be a set of rewrite rules and equations, and we let $R^=$ be the set of equations generated by changing all rules to equations. Suppose we have the following system R:

$$c \rightarrow b$$

$$f(x) \rightarrow b$$

$$g(x) \rightarrow a$$

$$f(x) = g(y)$$

$$e \rightarrow a$$

$$e \rightarrow d$$

For this system, $R^=$ would be

$$c = b$$

$$f(x) = b$$

$$g(x) = a$$

$$f(x) = g(y)$$

$$e = a$$

$$e = d$$

In general, given such a set R and an equation $s = t$, we are interested in whether $R^= \models s = t$. Note that by Birkhoff's theorem, if $R^= \models s = t$ then there is a proof of $s = t$ using equality replacements and rewriting in the forward and backward directions. For example, for the above system R, $c = b$ has this proof: $c \rightarrow b \leftarrow f(x) = g(y) \rightarrow a \leftarrow e \rightarrow d$. This proof has the proof trace $\rightarrow\leftarrow=\rightarrow\leftarrow\rightarrow$, showing how $=$ in proof patterns corresponds to the use of equations. As before, we are interested in defining a notion of completion and observing its effect on this proof. The equation $f(x) = g(y)$ cannot be oriented into a rule in either direction since it would lead to nontermination (since each side has a variable not found on the other side). For example, if we orient it as $f(x) \rightarrow g(y)$ then we have $f(x) \rightarrow g(f(x)) \rightarrow g(g(f(x))) \rightarrow ...$ since the y in $g(y)$ can be replaced by $f(x)$. Such an equation could be in the system originally, or it might be produced as a critical pair between the rules $h(x,y) \rightarrow f(x)$ and $h(x,y) \rightarrow g(y)$ or other rules. The idea of unfailing completion is to modify completion so that if a critical pair (s,t) cannot be oriented to preserve termination, the equation $s = t$ is added to the system instead of a rewrite rule. Then, critical pairs can be formed between this equation and the rewrite rules. In general, if (s,t) is a critical pair for the rewrite rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$, then (s,t) is also a critical pair for the equation $l_1 = r_1$ and the rule $l_2 \rightarrow r_2$, and (s,t) is also a critical pair fot the rule $l_1 \rightarrow r_1$ and the equation $l_2 = r_2$, and (s,t) is also a critical pair for the equations $l_1 = r_1$ and $l_2 = r_2$. In addition, the equation $l_1 = r_1$ and the equation $r_1 = l_1$ are not distinguished from each other. Thus, equations can be used in place of rewrite rules in unfailing completion, and equations can be oriented in either direction. The effect of these additional rules is to eliminate rewrite sub-proofs of the form $=\rightarrow$ and $\leftarrow=$ as well as eliminating sub-proofs of the form $\leftarrow\rightarrow$ by the usual critical pair completion operation. To prove that these sub-proofs can be eliminated, it is necessary to use a proof ordering similar to that used in [BDP89].

For example, using the equation $f(x) = g(y)$ and the rule $f(x) \rightarrow b$ we obtain the critical pair $(g(y), b)$ which can be oriented as $g(y) \rightarrow b$. Using the equation $f(x) = g(y)$ and the rule

$g(y) \rightarrow a$ we can obtain the critical pair $(f(x), a)$ which can be oriented as the rule $f(x) \rightarrow a$. Using the equation $f(x) = g(y)$ and itself we obtain $g(x) = g(y)$, which does not help in this example. From the rules $f(x) \rightarrow a$ and $f(x) \rightarrow b$ we obtain the critical pair (a, b) which can be oriented as $b \rightarrow a$. The rule $d \rightarrow a$ can be generated from $e \rightarrow a$ and $e \rightarrow d$ as before. The effect is to replace the rewrite sequence $c \rightarrow b \leftarrow f(x) = g(y) \rightarrow a \leftarrow e \rightarrow d$ with the sequence $c \rightarrow b \rightarrow a \leftarrow d$, as before. In this case a "valley" proof is obtained, but in general the form of the proof will be $\rightarrow\rightarrow\rightarrow \ldots \rightarrow=== \ldots \leftarrow \ldots \leftarrow\leftarrow\leftarrow$, that is, $\rightarrow^*=^*\leftarrow^*$. This means that if $s = t$ is in the equational theory of R, then s and t rewrite (using the completed R, which may be infinite) to s' and t', respectively, such that $s' = t'$ is a consequence of equations in the completed R. Therefore, after finitely many completion steps, enough rules and equations will be generated to prove $s = t$ using equations only. We have to be careful that the rewrite proof between s and t does not get longer and longer without limit, as the completion proceeds; to deal with this, we require that the termination ordering used not be changed during the completion (though it can be strengthened). For example, when a new equation $u = v$ is generated, we may strengthen the ordering so that u and v are comparable, to minimize the number of equations generated. Thus the termination ordering used may become stronger and stronger. For a discussion of unfailing completion, see [BDP89, BDH86]. Note that unfailing completion may loop or succeed (stop with a finite set of rules and equations) but it cannot fail.

We now present some inference rules that may be used in different versions of completion and give some results about them. The rule $CP(\rightarrow, \rightarrow)$ is the operation of forming critical pairs between rewrite rules and orienting them as new rewrite rules. The rule $CP(\rightarrow, =)$ (equivalently $CP(=, \rightarrow)$) is the operation of forming a critical pair between a rule and an equation. The rule $CP(=, =)$ is the operation of forming a critical pair between an equation and an equation. Adding a U prefix (for "unfailing") means that if the resulting critical pair (s, t) cannot be ordered, then an equation $s = t$ is added. Thus we have $UCP(\rightarrow, \rightarrow)$, $UCP(\rightarrow, =)$ et cetera. For all of these rules, if a critical pair is of the form (s, s) it is deleted and no new rule is added. We also have the rule $SIMP$, which simplifies existing rewrite rules using other rewrite rules. If a rule simplifies to something of the form $t \rightarrow t$, it is deleted. However, we don't allow the rewrite rule $s \rightarrow t$ to be applied to its own left-hand side, since that would cause the rule to become $t \rightarrow t$ and thus all rules would be deleted. Also, each time $SIMP$ is performed, we assume that rewrite rules are used in the order they are generated to simplify everything else. We can now describe completion as the set $\{CP(\rightarrow, \rightarrow), SIMP\}$ of inference rules, applied fairly, where $SIMP$ has precedence over CP. We can describe unfailing completion as $\{UCP(\rightarrow, \rightarrow), UCP(=, \rightarrow), UCP(=, =), SIMP\}$ with similar fairness and priority assumptions. Then we can show the following results; see [Hue81] and [BDP89] for similar results. For the following results, we assume that the ordering $>$ is total on ground terms. (Recall that a ground term is a term without variables.)

Theorem 11.1.1. *Suppose R is a set of rewrite rules and equations and $R^= \models s = t$. Fair unfailing completion starting from R and using the same ordering throughout (or one of increasing strength) and using the inference rules $\{UCP(\rightarrow, \rightarrow), UCP(=, \rightarrow), UCP(=, =), SIMP\}$ will eventually produce a proof of $s = t$ of form $\rightarrow^*=^*\leftarrow^*$.*

Here we don't have a confluence result, since R contains equations and rules. However, if s and t are continually rewritten to normal form during the completion process, then they will eventually be rewritten to s' and t' such that $s' = t'$ has an equational proof from S, where S is the set of generated rules and equations. The reason for this is that the ordering $>$ is well-founded. Therefore, normal forms s' and t' of s and t using S must eventually be reached that can never again be rewritten. But there is still a proof of $s' = t'$ in S of form $\rightarrow^*=^*\leftarrow^*$, by the above theorem. Therefore this proof has to be of the form $=^*$. In fact, there will be such a proof that is an "equational valley proof," that is, the instances of the terms used get smaller in the ordering and then larger. Possibly such equational proofs can be found by E-unification using the set E of equations in S.

11.2 Completion with inequations

A further modification that previews the use of equality in first-order theorem proving is to allow inequations in R. These results make use of the fact that $R \models s = t$ iff $R^= \cup \{s' \neq t'\}$ is unsatisfiable, where s' and t' are s and t with variables replaced by distinct new individual constants in a systematic way. These constants are called Skolem constants. For a discussion of their use in mechanical theorem proving, see [CL73, Lov78]. For example, $R^= \models f(x,y) = g(y,x)$ iff $R^=$ together with $f(a,b) \neq g(b,a)$ is unsatisfiable, where a and b are new constants. This technique permits us to define inference rules that operate directly on the goal $s = t$. Also, if R is a set of equations and rewrite rules and W is a set of inequations, then $R^= \cup W$ is unsatisfiable iff there is some inequation $s \neq t$ in W such that $R^= \cup \{s \neq t\}$ is unsatisfiable. Using this idea, we can treat any combination of rewrite rules, equations, and ground inequations. For this, we define $R^=$ as R with all rewrite rules replaced by equations; $R^=$ will contain inequations if R does. We define critical pairs with inequations in this way: If (s,t) is a critical pair for the rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ and the critical subterm occurrence is in l_2 then (s,t) is also a critical pair for the rule $l_1 \rightarrow r_1$ and the inequation $l_2 \neq r_2$. However, in this case the critical pair is made into an inequation $s \neq t$ rather than an equation or a rewrite rule. For example, $(f(a), g(c))$ is a critical pair for $f(c) \rightarrow f(a)$ and $f(x) \neq g(x)$, and this critical pair would be converted into the inequation $f(a) \neq g(c)$. The inference rule $CP(\neq, \rightarrow)$ performs this operation. Similarly, we can have critical pairs between inequations and equations; the inference rule $CP(\neq, =)$ performs this operation. Note that the U prefix is not used for inequations since the result of the rule is always an inequation. Finally, the rule $CONTR$ derives $FALSE$ if an inequation of the form $s \neq s$ is derived. It turns out that the rule $CP(\neq, \rightarrow)$ is not needed for ground inequations, but it is useful later. We then have the following results:

Theorem 11.2.1. *Fair unfailing completion starting from a set R of rules, equations, and ground inequations and using the inference rules $\{UCP(\rightarrow, \rightarrow)$, $UCP(=, \rightarrow)$, $UCP(=, =)$, $CP(\neq, =)$, $SIMP$, $CONTR\}$ will eventually generate $FALSE$ if $R^=$ is unsatisfiable.*

Proof. By an above remark, it suffices to consider the case where R contains only one ground inequation $s \neq t$. We know that we will generate a proof of $s = t$ of the form $\rightarrow^*=^*\leftarrow^*$. The rule $SIMP$ permits the elimination of the beginning and ending of the proof as the inequation is simplified (rewritten), leaving only the $=^*$ in the middle; this means that we can derive an inequation

$s' \neq t'$ such that a proof having pattern $=^*$ exists. Applications of the rule $CP(\neq, =)$ will give an equation of the form $t' \neq t'$. Then an application of $CONTR$ gives $FALSE$. Note that the rule $CP(\neq, \rightarrow)$ is not needed since any derived inequation will be a ground inequation, so that the effect of such a critical pair is obtained by rewriting the inequation. This result means that fair unfailing completion can be used as a complete theorem proving method for equational theories, that is, equational consequences can eventually be proved, but the procedure may run forever if it is asked to prove a non-consequence. ∎

11.3 Using equations as rewrite rules

There is still another variation on this idea. It is possible to make the simplification operation more powerful, permitting an equation to be used as a simplification in certain cases. This will have an application also in general theorem proving. This permits us to eliminate the inference rule $CP(\neq, =)$ in some theorems above. For this, we define the inference rule $SIMPO$ (ordered simplification) as follows:

If $A[s]$ is an equation, inequation, or rewrite rule containing the subterm s, and if $u = v$ is an equation having an instance $s = t$ such that $s > t$ (where $>$ is the termination ordering mentioned before), then replace $A[s]$ by $A[t]$. Thus, if $b * a > a * b$, an equation $f(a, b) = b * a$ can be simplified to $f(a, b) = a * b$ using the equation $x * y = y * x$. Note that the equation $x * y = y * x$ cannot be oriented, since it is symmetrical with respect to the left and right hand sides. However, its instance $b * a = a * b$ can be oriented and used to replace a subterm $b * a$ by $a * b$. There is a restriction on the rule $SIMPO$; namely, a rewrite rule cannot simplify itself. Otherwise, all rewrite rules would simplify to the form $s = s$, as mentioned above in the discussion of $SIMP$. As in the rule $SIMP$, we assume that equations are used in the order they are generated to simplify everything else; in fact, equations and rewrite rules are used together in the order they are generated, applying rules $SIMP$ and $SIMPO$ together.

We are still assuming that $>$ is total on ground terms, as before. There is a fairly strong result about completion methods using the inference rule $SIMPO$:

Theorem 11.3.1. *Fair unfailing completion starting from a set R of rules, equations, and ground inequations and using the inference rules $\{UCP(\rightarrow, \rightarrow), UCP(=, \rightarrow), UCP(=, =), SIMP, SIMPO, CONTR\}$ will eventually generate $FALSE$ if $R^=$ is unsatisfiable. Note that the inference rule $CP(\neq, =)$ is not needed for this.*

Proof. Suppose R is unsatisfiable. Then there is a ground inequation $s \neq t$ in R such that the rewrite rules and equations in R, together with $s \neq t$, are unsatisfiable. We can't directly use theorem 11.2.1 above, since the rule $SIMPO$ may cause the proof of form $\rightarrow^* =^* \leftarrow^*$ to be deleted. We know that there is some proof of $s = t$, by Birkhoff's theorem, in which rewrite rules and equations are both used, either in the forward or in the backward direction. Let the intermediate terms as in Birkhoff's theorem be $s_1 ... s_n$. Since s and t are ground terms, we can choose the s_i to be ground terms. Also, we can choose the s_i to be distinct since duplicates can be eliminated to obtain a shorter sequence. Then for each i, we have either $s_i > s_{i+1}$ or $s_i < s_{i+1}$ since the ordering is total on ground terms. The effect of the $UCP(=,=)$ operation on these ground terms is similar to the effect of the $CP(\rightarrow,\rightarrow)$ operation, namely, s_i that are larger than s_{i-1} and s_{i+1} can be eliminated from the equational proof. This is because if $s_i > s_{i+1}$ then we may have an equation with $s_i = s_{i+1}$ or $s_{i+1} = s_i$ as an instance, or we may have $s_i \rightarrow_R s_{i+1}$, but we cannot have $s_{i+1} \rightarrow_R s_i$ because the rewrite rules make terms smaller in the ordering $>$. Using the three UCP inference rules, such s_i may be eliminated from the proof. Eventually the sequence s_i becomes a valley in the sense that terms become progressively smaller along the sequence, then progressively larger. Then s and t may be simplified to the same thing (the middle of the valley) by rewriting or ordered simplification using the generated equations. Then an application of the rule $CONTR$ generates $FALSE$. This also gives a complete theorem proving procedure for equational theories. ∎

Ordered simplification is implemented in the ITP theorem prover of Argonne, now available as OTTER [McC89a]. The advantage is that it is not necessary to use a special unification algorithm for associative and commutative functions. A disadvantage is that it is often necessary to keep more terms than if an associative-commutative unification algorithm were used. Another system obtaining impressive results with ordered simplification is RRL [KZ91].

The idea of ordered simplification was taken further by Martin and Nipkow [MN90] and Peterson [Pet90]. The idea is to constrain rewrite rules by orderings on the terms replacing the variables. Thus we can have the rewrite rule $x * y \rightarrow y * x$ with the constraint that $x > y$ in the termination ordering. They present methods for showing that such constrained rewriting systems are ground confluent. For example, the following system reduces any ground term over the equational theory $\{(x * y) * z = x * (y * z), x * y = y * x\}$ to canonical form, if ordered simplification is done:

$$(x * y) * z \rightarrow x * (y * z)$$

$$x * y = y * x$$

$$x * (y * z) = y * (x * z)$$

Using this system, the term $(b * c) * a$ will rewrite to $b * (c * a)$ using $(x * y) * z \rightarrow x * (y * z)$, then to $b * (a * c)$ using ordered simplification and the equation $x * y = y * x$, then to $a * (b * c)$ using the equation $x * (y * z) = y * (x * z)$ and ordered simplification. Other systems like this that are canonical with respect to ordered simplification and rewriting are given in [MN90].

We now give oriented versions of the inference rules UCP. These are like the rules UCP except that they require that the instance of equations used have a certain property with respect to the termination ordering $>$, much like the rule $SIMPO$ above.

Definition 11.3.2. The rules $UCPO(=,=)$, $UCPO(=,\rightarrow)$, $CPO(=,\neq)$, and $CPO(\neq,\rightarrow)$ are like the rules $UCP(=,=)$, $UCP(=,\rightarrow)$, $CP(=,\neq)$, and $CP(\neq,\rightarrow)$, respectively, except for the following restriction: If an equation $r = s$ or an inequation $r \neq s$ is used in these rules and Θ is the critical substitution and the critical subterm occurrence is in r, then we cannot have $s\Theta > r\Theta$. Similarly, if the critical subterm occurrence is in s then we cannot have $r\Theta > s\Theta$.

Many of the preceding results still hold if the rules UCP et cetera are replaced by their ordered counterparts $UCPO$, et cetera. In fact, many of the results can be extended to inference rules in which inequations are similarly restricted. However, we think that the preceding results give a sufficient flavor of what can be done, and the reader may enjoy deriving some of the ordering results.

This framework can easily be extended to equation solving by a simple device. In equation solving, we are looking for an instance of s and t such that $R^= \models s = t$. This is like answer extraction in logic programming. For example, we may be looking for an x such that $append(x, [3, 4]) = [1, 2, 3, 4]$. To do this, we can consider R together with the rules $eq(s, t) \rightarrow FALSE$ and $eq(x, x) \rightarrow TRUE$, where "$eq$" is a new symbol; let this expanded R be R'. Then, some instance of $s = t$ is a consequence of R iff $R' \models TRUE = FALSE$. Note that $TRUE = FALSE$ is a ground equation, so the preceding results can be used to answer this question. Finally, from the resulting rewrite proof of $TRUE = FALSE$, we can extract the instance $s' = t'$ of $s = t$ such that $R \models s' = t'$. In this way, we essentially get the "narrowing" strategies of [Fay79, Hul80].

By casting the above results back in the original form, namely, by looking at what operations on R and $s = t$ correspond to the critical pair operations on R', we can obtain results about R directly. It turns out that many of the above results about sets of rewrite rules, equations, and ground inequations also hold for arbitrary sets of rewrite rules, equations, and possibly non-ground inequations. For example, we have the following result:

Theorem 11.3.3. *Fair unfailing completion starting from a set R of rules, equations, and possibly non-ground inequations and using the inference rules $\{UCP(\rightarrow, \rightarrow)$, $UCP(=, \rightarrow)$, $UCP(=, =)$, $CP(\neq, =)$, $CP(\neq, \rightarrow)$, $SIMP$, $CONTR\}$ will eventually generate $FALSE$ if $R^=$ is unsatisfiable.*

Proof. As before, it suffices to consider the case where R contains only one possibly non-ground inequation $s \neq t$. We transform a proof from $R \cup \{eq(s, t) \rightarrow FALSE, eq(x, x) \rightarrow TRUE, TRUE \neq FALSE\}$ as in theorem 11.3.1 into a proof of $FALSE$ from R. The inference rule $CP(\neq, \rightarrow)$ is needed now to simulate the effect of critical pairs between equations of the form $eq(u, v) \rightarrow FALSE$ and other rewrite rules. The inference rule $SIMPO$ is not needed (but won't hurt and is useful) because its effect on the inequation is simulated by the inference rule $CP(\neq, =)$. For this, we need to modify the rule $CONTR$ to derive $FALSE$ from an inequation that is unifiable with $x \neq x$. ∎

Even when completion runs forever, it generates a possibly infinite canonical set of rules. The set of rules is locally confluent, since any subproof of the form $\longleftrightarrow$ corresponds to a critical pair and is eliminated. The set of rules is terminating, because the rewrite ordering used is well-founded. Therefore the set of rules obtained is confluent. For unfailing completion, there may be equations and rules. Therefore we can't say that the possibly infinite set of rules and equations generated in the limit, is canonical. However, it turns out that using ordered simplification (the rule $SIMPO$), the limiting set of rules and equations is *ground confluent*, that is, any two ground terms s and t that are (R, E)-equivalent, will rewrite to a common term. Furthermore, one can show that if there is a finite canonical term rewriting system equivalent to (R, E), then unfailing completion using a suitable termination ordering, will eventually find it. This means that the completion procedure is also unfailing in the sense of never failing to find a finite canonical system, if one exists for the specified termination ordering.

11.4 Examples of completion

We now illustrate Knuth-Bendix completion on a common example. Suppose R is the following set of rules:

$$1 * x \to x$$

$$g(x) * x \to 1$$

$$(x * y) * z \to x * (y * z)$$

We can overlap the first or second rule with the third one. Suppose we rename variables in $1 * x \to x$, obtaining $1 * w \to w$, to avoid common variables, and then overlap overlap $1 * w$ with $(x * y) * z$, unifying $1 * w$ with the subterm $(x * y)$ This gives the term $(1 * y) * z$, which rewrites to $y * z$ using $1 * w \to w$ and to $1 * (y * z)$ using $(x * y) * z \to x * (y * z)$. Thus we have the critical pair $(y * z, 1 * (y * z))$. However, these terms both have $y * z$ as a normal form, since $1 * (y * z)$ rewrites to $y * z$ using the rule $1 * w \to w$. Next, we try overlapping $g(x) * x$ with $(x * y) * z$, renaming variables as before and then unifying $g(x) * x$ with the subterm $(x * y)$ and obtaining the term $(g(x) * x) * z$. This term rewrites to $1 * z$ using the rule $g(x) * x \to 1$ and to $g(x) * (x * z)$ using the rule $(x * y) * z \to x * (y * z)$. This gives us the critical pair $(g(x) * (x * z), 1 * z)$ which we rewrite to $(g(x) * (x * z), z)$ using the rule $1 * x \to x$ and orient into the rule $g(x) * (x * z) \to z$. Now we have four rules:

$$1 * x \to x$$

$$g(x) * x \to 1$$

$$(x * y) * z \to x * (y * z)$$

$$g(x) * (x * z) \to z$$

We will do another overlap. Let's consider rules 2 and 4, overlapping $g(x) * x$ with the subterm $(x * z)$ of $g(x) * (x * z)$. This gives the term $g(g(x)) * (g(x) * x)$. This term rewrites to $g(g(x)) * 1$ using the rule $g(x) * x \to 1$ and to x using the rule $g(x) * (x * z) \to z$. We thus obtain the critical pair $(g(g(x)) * 1, x)$ which we orient into the rule $g(g(x)) * 1 \to x$. We now have a total of five rules:

$$1 * x \to x$$

$$g(x) * x \to 1$$

$$(x * y) * z \to x * (y * z)$$

$$g(x) * (x * z) \rightarrow z$$

$$g(g(x)) * 1 \rightarrow x$$

This process continues, until a ten rule canonical term rewriting system is obtained if the correct termination ordering is used. During this process some rules that have been added are later deleted or simplified; for example, the rule $g(g(x)) * 1 \rightarrow x$ is eventually eliminated. We finally obtain

$$1 * x \rightarrow x$$

$$g(x) * x \rightarrow 1$$

$$(x * y) * z \rightarrow x * (y * z)$$

$$g(x) * (x * z) \rightarrow z$$

$$x * 1 \rightarrow x$$

$$g(1) \rightarrow 1$$

$$g(g(x)) \rightarrow x$$

$$x * g(x) \rightarrow 1$$

$$x * (g(x) * y) \rightarrow y$$

$$g(x * y) \rightarrow g(y) * g(x)$$

It is interesting in this case how quickly new and useful rules are derived.

As an example of unfailing completion, suppose R is the following set of rules:

$$(x * y) * z \rightarrow x * (y * z)$$

$$1 * x \rightarrow x$$

$$x * 1 \rightarrow x$$

$$x * x \rightarrow 1$$

We want to show that $R \models x * y = y * x$. A direct proof is $x * y \leftarrow x * (1 * y) \leftarrow x * (((x * y) * (x * y)) * y)...(x * x) * ((y * x) * (y * y)) \rightarrow 1 * ((y * x) * 1)...(y * x)$. Using unfailing completion, we can overlap the rules $(x * y) * z \rightarrow x * (y * z)$ and $x * x \rightarrow 1$, unifying $x * x$ with $(x * y) * z$ and obtaining the critical pair $((x * y) * (x * y), 1)$ and the rule $x * (y * (x * y)) \rightarrow 1$. Overlapping this rule with $(x * y) * z \rightarrow x * (y * z)$, we obtain the rule $x * (y * (x * (y * z))) \rightarrow z$. Overlapping the subterm $x * (y * z)$ of this rule with $x * (y * (x * y)) \rightarrow 1$, we obtain the pair $(x * (y * (x * (y * (x * y)))) \rightarrow x * y)$, but the left-hand side simplifies to $x * y$ so this is not productive. Overlapping the subterm $y * z$ of $x * (y * (x * (y * z)))$ with the rule $x * x \rightarrow 1$, we obtain $x * (y * x) \rightarrow y$. Overlapping this with the term $x * y$ of $(x * y) * z \rightarrow x * (y * z)$, we obtain $x * (y * (x * z)) \rightarrow y * z$. Finally, overlapping the subterm $x * z$ of this rule with the left-hand side of the rule $x * x \rightarrow 1$, we obtain the term $x * (y * (x * x))$ which rewrites to $x * y$ using $x * x \rightarrow e$ and to $y * x$ using $x * (y * (x * z)) \rightarrow y * z$, giving us the equation $x * y = y * x$. Notice that this equation cannot be oriented. The reader may notice some similarities between this derivation and the proof of $x * y = y * x$ given above. Of course, we could also have proved this by negating the conclusion and adding Skolem constants, obtaining the inequation $a * b \neq b * a$; one of the methods for showing unsatisfiability could then have been used.

We give a few more examples. Ternary Boolean algebra [Gra47] is defined by the rules and equation

$$f(y, x, x) \rightarrow x$$

$$f(x, y, g(y)) \rightarrow x$$

$$f(f(v, w, x), y, f(v, w, z)) = f(v, w, f(x, y, z))$$

¿From these many more can be derived, including the following:

$$f(x, x, y) \rightarrow x$$

$$f(g(x), x, y) \rightarrow y$$

$$f(x, y, x) \rightarrow x$$

$$g(g(x)) \rightarrow x$$

$$f(x, y, z) = f(z, y, x)$$

Combinatory logic is a well known undecidable system that can be expressed as a set of rewrite rules:

$$((S * x) * y) * z \rightarrow (x * z) * (y * z)$$

$$(K * x) * y \rightarrow x$$

$$I * x \rightarrow x$$

This system is also orthogonal, hence confluent. But it is not terminating.

11.5 Modifications and extensions to completion

We illustrate another phenomenon with the following single axiom for group theory [HN52]:

$$x/((((x/x)/y)/z)/(((x/x)/x)/z)) \rightarrow y$$

When this system is completed, a canonical term rewriting system is obtained. During the completion, the equation $x/x = y/y$ is obtained. This cannot be oriented. However, it is clear that the value of x/x does not depend on x, since x does not appear on the right-hand side. Therefore we introduce a new constant a and the rule $x/x \rightarrow a$. In general, if an equation $r = s$ is derived, and at least one variable appears on only one side of this equation, then we can add rules $r \rightarrow f(x_1 ... x_n)$ and $s \rightarrow f(x_1 ... x_n)$ where the x_i are all the variables in common to r and s and f is a new function symbol. Another idea that works well is to postpone the orientation of equations that cannot be made into rewrite rules, since rules may be generated later that permit these equations to be oriented or deleted.

We have not specified how to choose which critical pair to consider at any given time in completion or unfailing completion, except that it should be done fairly. One idea is to do some kind of breadth first search to insure that all possibilities are examined. Another good idea is to overlap the two smallest equations that have not been tried together so far. This is good because it tends to produce small rules, and small rules tend to be more useful than large rules since small rules apply more often. It might also be useful to look for small equations with the largest number of variables. Using small equations is fair since there are only finitely many equations of a given size or smaller. Thus, eventually all equations that can be generated within that size bound will be generated, and larger equations will be considered.

Another idea [Les84] is to run the completion procedure for a while, then filter out the most interesting equations (typically small equations), add them to the original set R, and repeat this process. This idea of "filter and reuse" tends to focus attention on the more interesting equations, and can produce good results. This even seems to be a useful idea for general first-order theorem provers, and the author has done some limited experiments with it.

There has been some theoretical work on restrictions of the critical pairs that still preserve completeness of Knuth-Bendix or unfailing completion [WB83, KMN88, ZK90]. This can save some work, but it does not seem that such methods improve running times by more than a small factor, except in the associative-commutative case where it can make a dramatic difference. Sometimes one can show that certain critical pairs are never needed, since they correspond to a sub-proof of the form $r \leftarrow s \rightarrow t$ for which other proofs exist not involving s. This may be the case, for example, if $s \rightarrow u$ for some term u other than r and t.

In order to perform completion, it is necessary to choose a termination ordering. Since many such orderings are known, this can be difficult to do. In [Pla86], a method is proposed to avoid this problem. There, nondeterministic versions of completion are given; the nondeterminism has to do with how a critical pair (s,t) is converted into a rule, whether $s \rightarrow t$ or $t \rightarrow s$ is chosen. The idea is to give a simple criterion for nontermination; for any system R satisfying this criterion, there is no simplification ordering $>$ such that for all rules $l \rightarrow r$ in R, $l > r$. When this happens, some nondeterministic choice has to be re-done, and completion is again attempted. This method is guaranteed to be able to generate any system that could be generated using a simplification ordering to orient the critical pairs.

It turns out that for a given termination ordering $>$, there is a unique completion of a given term rewriting set R. We say a set R of rewrite rules is *reduced* if all rules $l \rightarrow r$ in R are rewritten as much as possible with respect to R. Formally, l and r are in normal form with respect to $R - \{l \rightarrow r\}$. Some authors use the term *canonical* for this property. Now, if S_1 and S_2 are two canonical systems equivalent to R, and both are reduced, and for all rules $l \rightarrow r$ in $S_1 \cup S_2$, $l > r$, then S_1 and S_2 are identical, up to renaming of variables. For a proof, see [Met83]; the result was also contained in some unpublished work of Lankford and others. This result implies that the choice of an ordering determines the final result of completion.

H. Kirchner [Kir87] shows that in some cases, when completion does not terminate, patterns in the infinite set of rules generated may be observed and used to generate "meta-rules" that encode these patterns. In some case this permits a canonical term rewriting system to be constructed using "meta-term rewriting."

In [Sny89, GNP+88] a method is given that completes ground term rewriting systems in polynomial time.

As for applications of completion, Dershowitz [Der85] shows how completion can be applied to program generation. It also has an immediate application to equational theorem proving, as discussed already. Similar ideas also apply to more general theorem proving, as will be discussed in section 16. Also, a modified completion is an integral part of many of the inductionless induction theorem proving methods.

12 Rewriting strategies

There is often a choice which redex to rewrite during term rewriting. There are a number of systematic ways to choose among the redexes, with implications for the rewriting process. We now discuss some of these methods. First we discuss some possibly new results that apply to general systems. Then we discuss some known results about orthogonal systems.

We first discuss the decidability of confluence of two restricted rewriting strategies. We define *innermost rewriting* to be rewriting in which a minimal redex is always chosen. That is, the redex chosen must be a subterm occurrence which has no proper subterms that are reducible. We define *outermost rewriting* to be rewriting in which a maximal redex is always chosen, that is, a subterm occurrence that is not properly contained in any other redexes. We say a term rewriting system is *innermost confluent* if the innermost rewriting relation is confluent, and it is *outermost confluent* if the outermost rewriting relation is confluent. Note that confluence implies innermost confluence and outermost confluence but a system can be innermost or outermost confluent without being confluent. Innermost confluence is interesting since denotational semantics is often defined by a kind of innermost evaluation procedure, which is similar to innermost rewriting. Outermost confluence is interesting because outermost reduction corresponds to lazy evaluation. If a system is outermost confluent, that means that the lazy semantics is well-defined, regardless of the order in which outermost reductions are done and regardless of which rules are applied. Also, nonterminating outermost confluent systems are those for which a semantics based on infinite terms seems most natural, since lazy evaluation for nonterminating programs can lead to infinite structures. However, we are not sure what the implications of innermost and outermost confluence are for decision procedures based on rewriting.

Theorem 12.0.1. *Innermost confluence is decidable for terminating rewrite systems.*

Proof. The only way that a term can have two innermost reductions is if the two redexes are disjoint or identical. Disjoint redexes do not cause a problem, but can be treated as in the critical pair theorem, theorem 10.0.2. Identical redexes correspond to "overlays," that is, critical pairs in which the critical subterm occurrence is the left hand side of some other rule. Formally, if $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ are two different rules in R and l_1 and l_2 unify with most general unifier Θ, then $(r_1\Theta, r_2\Theta)$ is an *overlay* for the rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$. As in the critical pair lemma, we can show that R is innermost confluent iff for all overlays (s, t), s and t reduce to a common term by innermost rewriting. ∎

In fact, this result can be partially extended to nonterminating systems, as follows:

Theorem 12.0.2. *Suppose R has no overlays, that is, R is non-overlapping. Then R is innermost confluent.*

Proof. The innermost rewriting relation is strongly confluent in that case, hence confluent. ∎

In fact, we can extend this to show that if for all overlays (r, s) either $r \rightarrow s$ or $s \rightarrow r$ or for some t, $r \rightarrow t$ and $s \rightarrow t$, then R is innermost confluent. The proof is as above, namely, that innermost rewriting is strongly confluent in that case.

The situation with outermost rewriting is more complicated. The reason is that one reduction can create a redex containing the rewritten term, enabling a new reduction that can prevent other reductions from occurring.

Theorem 12.0.3. *Outermost confluence is undecidable, even for terminating systems.*

Proof. We sketch the proof, omitting some technical details. Consider the following system:

$$a \rightarrow m(s_0, b, b)$$

$$a \rightarrow LOOP$$

$$f(m(x, y, z)) \rightarrow m(x', y', z')$$

$$m(s, y, z) \rightarrow LOOP \ for \ non - accepting \ states \ s$$

$$m(s_H, y, z) \rightarrow HALT$$

$$f(HALT) \rightarrow HALT$$

$$f(LOOP) \rightarrow LOOP$$

Here $m(x, y, z)$ represents a Turing machine configuration where x is the state, y is the tape to the left of the state, and z is the tape to the right of the state. Also, b represents an infinite blank tape. Thus, we start with the tape blank in both directions. Also, in the rule $f(m(x, y, z)) \rightarrow m(x', y', z')$, it is understood that the configuration (x', y', z') is the result of one legal move from (x, y, z). The first rule is for the start state s_0. The fourth rule is for non-accepting states s. The fifth rule is for the halt state s_H. We omit most details here, but a sequence of tape symbols can be represented as a composition of unary function symbols and the representation of legal moves by term rewriting systems is standard in the literature.

The idea is that $f^i(m(x, y, z))$ reduces to a configuration obtained by i moves from $m(x, y, z)$. Each step consumes an outer enclosing f. When all such f have been consumed, then the rule $m(s, y, z) \rightarrow LOOP$ applies if s is not an accepting state. This is where we make use of outermost reduction: the rule $m(s, y, z) \rightarrow LOOP$ does not apply until all outer enclosing f have been consumed. Therefore $f^i(m(s_0, b, b))$ will always reduce to $LOOP$ unless the Turing machine halts on blank tape, in which case for some i this will reduce to $HALT$. Since a reduces to both $m(s_0, b, b)$ and $LOOP$, the system will be outermost confluent iff the Turing machine does not halt on blank tape. Since the blank tape halting problem is undecidable, outermost confluence is also undecidable.

Note that the system given is left-linear. Also, note that the system is typically not confluent: let (r, s, t) be some configuration such that r is not the halt state but in one move the Turing machine can move into a halt state in a configuration (r', s', t'). (If such (r, s, t) does not exists, then the Turing machine cannot halt.) Then $f(m(r, s, t)) \rightarrow m(r', s', t') \rightarrow HALT$ and also $f(m(r, s, t)) \rightarrow f(LOOP) \rightarrow LOOP$. However, the reduction $f(m(r, s, t)) \rightarrow f(LOOP)$ is not outermost, so the system can still be outermost confluent, even though it is not confluent in this case. ■

In general, it is possible to show that for left-linear term rewriting systems, if there are no overlays and if there are no overlaps between right-hand sides and proper subterms of left-hand sides of rules, the system is outermost confluent. Such overlaps correspond to cases in which the application of a rule can enable the application of another rule at an enclosing redex.

We now discuss orthogonal systems. Not much is known about reduction strategies for non-orthogonal systems; the above results, however, do apply to non-orthogonal systems. We discuss five of the most common strategies: leftmost innermost, parallel innermost, leftmost outermost, parallel outermost, and full substitution rule reduction. Leftmost innermost reduction is the strategy in which the leftmost among the innermost redexes is always chosen. Parallel innermost reduction is the strategy in which all innermost redexes are rewritten at once. Leftmost outermost reduction is the strategy in which the leftmost among all the outermost redexes are rewritten. Parallel outermost reduction is the strategy in which all outermost redexes are simultaneously rewritten. Full substitution rule reduction is the method in which all redexes are rewritten at once. This requires a little care to explain properly, but the idea should be clear.

We now give some examples of the various strategies:

Suppose R is the following system for addition:

$$plus(0, x) \rightarrow x$$

$$plus(s(x), y) \rightarrow s(plus(x, y))$$

Consider the term $f(plus(0, s(0)), plus(0, plus(s(0), s(s(0)))))$. Using leftmost outermost reduction, we would choose the redex $plus(0, s(0))$. Using leftmost innermost reduction, the same redex would be chosen. Using parallel innermost reduction, the redexes $plus(0, s(0))$ and $plus(s(0), s(s(0)))$ would be chosen. Using parallel outermost reduction, the redexes $plus(0, s(0))$ and $plus(0, plus(s(0), s(s(0))))$ would be chosen. Using full substitution, all three redexes would be rewritten at once.

We say a strategy is *normalizing* if, given any term t which has a normal form, the strategy will rewrite t to a normal form. A strategy is *perpetual* if, given a term with at least one infinite rewrite sequence, the strategy will not terminate (thus the strategy avoids normal forms). It is known that the full substitution rule and parallel outermost rewriting are normalizing; in fact, the full substitution rule has an even stronger "cofinal" property that is discussed in [Klo91]. The idea of cofinal is that any rewriting sequence can be extended to yield a term obtainable by the specified (full substitution rule) strategy. However, leftmost innermost reduction and parallel innermost reduction are perpetual. Certain restricted classes of systems are known with different properties. For example, an orthogonal system is *left normal* if all the function symbols and individual constant symbols occur to the left of the variables in the left-hand sides of the rules. Thus a rule $f(a, X) \rightarrow \ldots$ is left normal but $f(X, a) \rightarrow \ldots$ is not. For left normal systems, leftmost outermost reduction is normalizing. In fact, for such systems, one can show that if s rewrites to t, then there is a rewrite sequence from s to t in which the rewriting goes from left to right in the term, and it is never necessary to rewrite subterms to the left of a redex chosen for rewriting. It would be interesting to extend some of these results to non-orthogonal systems. For more such results about orthogonal systems, see [Klo91].

We are also interested in the efficiency of reduction. We would like to rewrite only those redexes that must be rewritten to reach a normal form. Furthermore, we want to be able to compute which redexes to rewrite. A reduction strategy, in general, is a mapping from terms to terms, so that if s maps to t, then s rewrites to t in one step. If s and t are identical, then s must be in normal form. Such a strategy gives a method of deciding which rewrites to perform. It turns out that, under fairly general conditions, it is possible to compute which reductions to perform so as to reach a normal form. However, such a general result turns out to be of little practical usefulness because of its complexity. Furthermore, reducing all outermost redexes in parallel is normalizing, as stated above, and is also easily computable. In addition, we will mention later that at least one outermost redex is needed for each term. So parallel outermost rewriting is efficient, in the sense that one of the redexes used at each step is needed. However, this is not a one-step strategy because more than one redex is rewritten at each step. Still, this strategy is good enough that the concept of computable one-step reduction strategies does not seem of major importance unless there are a large number of outermost redexes. It turns out that there is a restricted class of term rewriting systems for which it is possible to compute a needed redex at each step and obtain a normalizing reduction strategy. These are the *strongly sequential* systems, which we discuss next. First we present some results about needed redexes.

13 Strong sequentiality

In the discussion of strong sequentiality, we assume that all term rewriting systems are orthogonal. See section 10.1 for a definition of orthogonality.

Suppose $s \to t$. We now define a relation of *descendent* between subterm occurrences of s and t. The idea is that if $s \to t$, then there is a relationship between subterm occurrences in s and t; some subterm occurrences in t come from certain subterm occurrences in s. The descendent relation keeps track of which subterm occurrences in s are responsible for creating which subterm occurrences in t. We can think of it this way: Suppose one of the subterm occurrences in s were marked, by changing the function symbol to a new symbol of the same arity; where would this new symbol occur in t? Formally, suppose $l \to r$ is a rewrite rule and s and t are terms and $\overline{\overline{s}} = \alpha\overline{\overline{l\theta}}\beta$ and $\overline{\overline{t}} = \alpha\overline{\overline{r\theta}}\beta$; thus $s \to t$. Let s' be s with an occurrence of a subterm $f(u_1 ... u_m)$ replaced by $f'(u_1 ... u_m)$. Let t' be t with an occurrence of $f(v_1 ... v_m)$ replaced by $f'(v_1 ... v_m)$. We want to state under what conditions the marked subterm in t' is a descendent of the marked subterm in s'. We know that $\overline{\overline{s'}} = \alpha'\overline{\overline{l\theta'}}\beta'$, where the primes indicate that possibly one symbol is marked. Let t'' be such that $\overline{\overline{t''}} = \alpha'\overline{\overline{r\theta'}}\beta'$. Now, $s' \to t''$ and t'' has one or more subterms marked. (There may be more than one if r contains repeated variables.) We say that the marked subterm of t' is a descendent of the marked subterm of s' if t'' can be obtained from t' by eliminating all but one of the marks. We may further rewrite t to u. For derivations having more than one rewrite, we can say a subterm occurrence in u is a descendent of one in s if there is some subterm occurrence in t that is a descendent of the occurrence in s and having the occurrence in u as a descendent. A similar marking idea was used in [Middeldorp and Toyama, 1991]. The following definition formalizes this idea.

Recall that a subterm occurrence β of a term s such that $\overline{\overline{s}}$ is $\alpha\beta\gamma$ can be represented by a triple (α, β, γ). Recall also that we use hats to indicate instances, so $\hat{u}$ is an instance of u. Corresponding to a rewrite sequence, we construct a rewrite sequence for subterm occurrences, where each step in the rewrite sequence is converted to a rewrite step on subterm occurrences as follows:

We say a redex is *contracted* if it is chosen for rewriting. Suppose $s \to t$ and (α, β, γ) is a subterm occurrence in s. (Thus $\overline{\overline{s}}$ is $\alpha\beta\gamma$.) Suppose the contracted redex of s is in α. Then we write $(\alpha, \beta, \gamma) \to (\alpha_1, \beta, \gamma)$, where $\overline{\overline{t}}$ is $\alpha_1\beta\gamma$. If the contracted redex of s is in γ, we write $(\alpha, \beta, \gamma) \to (\alpha, \beta, \gamma_1)$, where $\overline{\overline{t}}$ is $\alpha\beta\gamma_1$. If the contracted redex of s is a proper subterm of β, then we write $(\alpha, \beta, \gamma) \to (\alpha, \beta_1, \gamma)$, where $\overline{\overline{t}}$ is $\alpha\beta_1\gamma$.

Suppose $l \to r$ is the rewrite rule used to rewrite s to t, and let Θ be the substitution applied to it. Now, let's select out an occurrence of a variable x in l, assuming that x is not l. We assume l is linear for this. So $\overline{\overline{l}}$ is $l_1 x l_2$. Let's also select out an occurrence of x in r, if it exists. There may be more than one. So $\overline{\overline{r}}$ is $r_1 x r_2$. Suppose $x\Theta$ is u. Then $\overline{\overline{l\Theta}}$ is $\hat{l_1}\overline{\overline{u}}\hat{l_2}$ and $\overline{\overline{r\Theta}}$ is $\hat{r_1}\overline{\overline{u}}\hat{r_2}$. Thus $\overline{\overline{s}}$ is $\alpha\hat{l_1}\overline{\overline{u}}\hat{l_2}\beta$ and $\overline{\overline{t}}$ is $\alpha\hat{r_1}\overline{\overline{u}}\hat{r_2}\beta$. Suppose moreover that $\overline{\overline{u}}$ is $u_1\delta u_2$, where δ is a subterm occurrence and u_1 and u_2 may be empty. Then the indicated occurrence of δ in $\overline{\overline{s}}$ rewrites to the indicated occurrence of δ in $\overline{\overline{t}}$. More precisely, $(\alpha\hat{l_1}u_1, \delta, u_2\hat{l_2}\beta) \to (\alpha\hat{r_1}u_1, \delta, u_2\hat{r_2}\beta)$. Note that if a variable is repeated on the right-hand side, there may be more than one occurrence in t that an occurrence in s rewrites to.

Finally, if $(\alpha_1, \beta_1, \gamma_1) \to^* (\alpha_n, \beta_n, \gamma_n)$ then we say that the subterm occurrence $(\alpha_n, \beta_n, \gamma_n)$ is a *descendent* of $(\alpha_1, \beta_1, \gamma_1)$. For example, suppose we rewrite s which is $f(g(a), b)$ to t which is $f(h(a, a), b)$ using the rule $g(x) \to h(x, x)$. Then t is a descendent of s, b in t is a descendent of b in s (formally, $(fga, b,) \to (fhaa, b,))$, and both occurrences of a in t are descendents of the occurrence of a in s.

We say that a subterm occurrence is *needed* in a reduction to normal form if at least one descendent of the subterm is contracted in the reduction. A subterm occurrence is *needed* for a term having a normal form if it is needed in every derivation of the normal form from the term. Note that it is difficult to find needed redexes using this definition, since it is necessary to consider all reductions to normal form. The following results are known:

Every term that has a normal form, contains a needed redex. It is sufficient to contract needed redexes to find the normal form, and any sequence of contractions of needed redexes will eventually produce a normal form (for arbitrary orthogonal systems). So, there is always an "optimal" reduction strategy for any term using an orthogonal term rewriting system. The idea of sequentiality and strong sequentiality is to give a class of systems such that it is easy to find such a needed redex. Also, it is known that some outermost redex is needed, in general. But we don't know which one.

We mention that the concept of a sequential term rewriting system was discussed in [HL79] and [Klo91, KM91]. However, for simplicity we only discuss strong sequentiality, which is more useful. We actually use a modified definition by Klop [Klo91, KM91], which they assert is equivalent to the original. First, we need to define a context; this is just a term with "holes" in it, represented by $\Box$. Each occurrence of $\Box$ is a place in which an arbitrary term can be substituted. Such terms can be rewritten, as usual. A context that is irreducible is said to be in normal form. Suppose s is a context, and $\overline{\overline{s}}$ is $\alpha_1\Box\alpha_2\Box...\Box\alpha_n$, where none of the α_i contain $\Box$. Then we say that a term s' with $\overline{\overline{s'}} = \alpha_1\overline{\overline{t_1}}\alpha_2\overline{\overline{t_2}}...\overline{\overline{t_{n-1}}}\alpha_n$ is an *instance* of the context, for arbitrary terms t_i. For example, $f(a, g(b))$ is an instance of $f(\Box, \Box)$. We now define the rewriting relationship $\to ?$, which, roughly speaking, means that a redex can be replaced by an arbitrary term.

Definition 13.0.1. Suppose $l \to r$ is a rewrite rule in R and s is a term and $\overline{\overline{s}}$ is $\alpha_1\overline{\overline{u}}\alpha_2$, where u is the contracted redex. Suppose $l\Theta$ is u. Suppose t is an arbitrary term. Let v be a term such that $\overline{\overline{v}}$ is $\alpha_1\overline{\overline{t}}\alpha_2$. Then we say $s \to ?_R v$. The idea is that the $\to ?_R$ relation ignores right-hand sides and replaces redexes by arbitrary terms. For example, if R contains $g(a) \to c$ then $f(b, g(a)) \to ?_R f(b, t)$ for any term t. Usually we omit the R subscript on $\to ?$.

Definition 13.0.2. A redex of a term s is *strongly needed* for R if some descendent of s is contracted in every $\to ?_R$ reduction of s to normal form (or if s has no normal form).

Definition 13.0.3. A term rewriting system R is *strongly sequential* if for any context s in normal form, some occurrence of $\Box$ in s is strongly needed in $\to ?$ reduction. More precisely, suppose $\overline{\overline{s}}$ is $\alpha_1\Box\alpha_2\Box...\Box\alpha_n$, where none of the α_i contain $\Box$. For each instance s' of s with $\overline{\overline{s'}} = \alpha_1\overline{\overline{t_1}}\alpha_2\overline{\overline{t_2}}...\overline{\overline{t_{n-1}}}\alpha_n$ and all t_i redexes, we can reduce this instance to normal form using $\to ?$ reductions. A redex in s'

that is needed in all such reductions to normal form, is said to be ***strongly needed*** for $\rightarrow ?$ reduction. We say R is *strongly sequential* if for all such contexts s in normal form, there is an i such that for all instances s' with $\overline{\overline{s'}} = \alpha_1 \overline{\overline{t_1}} \alpha_2 \overline{\overline{t_2}} ... \overline{\overline{t_{n-1}}} \alpha_n$, t_i is strongly needed for $\rightarrow ?$ reduction. The idea is that the choice of which subterm t_i is strongly needed, only depends on the structure of the context, and not on the terms t_i themselves.

Now, all of the subterms t_i can be reduced. By orthogonality of R, it follows that the redexes t_i are outermost redexes, since s is in normal form with respect to ordinary rewriting. If R is strongly sequential, then at least one of these outermost redexes t_i needs to be reduced in any reduction of s' to normal form. Thus in any term, we can pick one of the outermost redexes that must be reduced, and reducing such redexes eventually gives a normal form. This means that the context s is "strict" in this redex, in a sense; the normal form of s really depends on the normal form for t_i. For example, suppose $R = \{f(b, b) \rightarrow c,\ a \rightarrow b\}$. Consider the context $f(\square, \square)$ and an instance $f(s, t)$ where s and t are redexes. This means that s and t are either a or $f(b, b)$, considering R. In all cases, both s and t have to be reduced to normal form in order to obtain the normal form of $f(s, t)$, even using $\rightarrow ?$ reduction. Therefore we can pick the first $\square$ in $f(\square, \square)$ as strongly needed. In order to show that R is strongly sequential, we also have to consider other normal form contexts, such as $f(f(\square, \square), \square)$, $f(b, f(\square, \square))$, et cetera.

It turns out that the property of strong sequentiality is decidable, although the decision procedure, from [HL79] and [Klo91], is quite complex. We will not discuss this decision procedure. Instead, we will give a simple method for finding strongly needed redexes for $\rightarrow ?$ reductions in strongly sequential systems. Note that any redex that is strongly needed for $\rightarrow ?$ reduction, is also strongly needed for ordinary $\rightarrow$ reduction. To find strongly needed redexes in terms (not contexts), note that one of the maximal redexes must be needed in any orthogonal system. It turns out that the maximal redexes are the ones of interest for strongly sequential systems. Suppose $\overline{\overline{s'}}$ is $\alpha_1 \overline{\overline{t_1}} \alpha_2 \overline{\overline{t_2}} ... \overline{\overline{t_n}} \alpha_{n+1}$, where the t_i are the maximal (outermost) redexes. Consider the context s with $\overline{\overline{s}} = \alpha_1 \square \alpha_2 \square ... \square \alpha_{n+1}$. We know that this context is in normal form since the t_i are outermost redexes. Therefore, if the system R is strongly sequential, there is an i such that for all such instances s', t_i is strongly needed for $\rightarrow ?$ reduction. Therefore, for this particular instance s', t_i is strongly needed for $\rightarrow ?$ reduction. Therefore, for this particular instance s', t_i is strongly needed for ordinary $\rightarrow$ reduction.

We still need a way to find the i such that t_i is strongly needed. The corresponding (i-th) occurrence of $\square$ in the context s is called an $nf?$ index in [Klo91], and we will use this terminology. First we present the concept of $\square$ reductions, instead of the Ω reductions presented in [Klo91].

Suppose s is a context and $\overline{\overline{s}}$ is $\alpha\beta\gamma$ and $l \rightarrow r$ is a rule. Suppose there is a context u that is an instance of β and of l. It is an instance of s obtained by replacing different occurrences of $\square$ by possibly different terms, but variables are not replaced. Also, u is an instance of l obtained by replacing variables by terms, different occurrences of a variable being replaced by the same term. So u is a kind of unifier of β and l except that a different notion of substitution is used for β and l. Then we say that the context $\alpha\square\gamma$ is obtained from $\alpha\beta\gamma$ by $\square$ rewriting. For example, suppose R contains the rule $f(x, a) \rightarrow b$. Then the context $g(f(c, \square))$ rewrites to $g(\square)$ by $\square$ rewriting. Note that the right-hand sides of rules don't affect the result.

Let $\Box(t)$ be the normal form of t obtained by $\Box$ reduction. This is well defined, since $\Box$ reduction is confluent and terminating. We call $\Box(t)$ the $\Box$ *normal form* of t. One can then show that an occurrence of $\Box$ in a context s is an $nf?$ index if the $\Box$ normal form of s, with the specified occurrence of $\Box$ replaced by a new symbol p, contains p. Then, as mentioned above, we can find strongly needed redexes for arbitrary reducible terms using this technique.

To efficiently find needed redexes of a term s' for strongly sequential systems, we first replace all maximal redexes by $\Box$, obtaining context s. We then compute the $\Box$ normal forms of all subterms of s. This can be done in time proportional to the size of the term by innermost reduction. Then, to see if an occurrence of $\Box$ in s is an $nf?$ index, we replace it by a new symbol p, and replace all maximal disjoint terms of s by their $\Box$ normal forms. Then p is needed iff the resulting term is $\Box$ irreducible. For this we only have to check redexes properly containing p; the number of such is bounded by the depth of the term.

Note that if one rewrite is done, then most subterms have not changed, so their $\Box$ normal forms have not changed. Therefore the recomputation can be done efficiently to find strongly needed redexes in the new term.

This method finds a needed redex, but doesn't completely address efficiency. We may get duplication of effort if a redex has two descendents i.e. $f(x) \rightarrow g(x,x)$ and both x have to be reduced. This can be helped by some kind of directed acyclic graph representation. Thus we obtain graph reduction, essentially.

As an example, consider the following system from [HL79]:

$$f(g(a,x), f(b,y)) \rightarrow s_1$$

$$f(g(x,a), f(c,y)) \rightarrow s_2$$

$$g(d,d) \rightarrow s_3$$

where s_1, s_2, and s_3 are arbitrary. For this system we get the $\Box$ reductions

$$f(g(a,x), f(b,y)) \rightarrow \Box$$

$$f(g(x,a), f(c,y)) \rightarrow \Box$$

$$g(d,d) \rightarrow \Box$$

plus other reductions obtained by replacing proper subterms of the left-hand sides by $\Box$, such as

$$f(g(\Box,x), f(b,y)) \rightarrow \Box$$

$$f(\Box, f(b,y)) \rightarrow \Box$$

$$f(g(\Box,x), f(\Box,y)) \rightarrow \Box$$

$$\ldots$$

Consider the context $f(g(\square,\square),f(g(\square,\square),\square))$. We show that no occurrence of $\square$ is needed. Let's replace all but the first one by $\square$, and the first one by p. We get the term

$$f(g(p,\square),f(g(\square,\square),\square))$$

By $\square$ rewriting, we get successively the following terms:

$$f(g(p,\square),f(\square,\square))$$

(using $g(d,d) \to \square$)

$$\square$$

(using $f(g(x,a),f(c,y)) \to \square$).

Since p does not occur in the result, the first occurrence of $\square$ is not strongly needed. In a similar way, one can show that no occurrence of $\square$ is strongly needed. Thus this system is not strongly sequential.

We give an example of a strongly sequential system:

$$plus(0,x) \to x$$

$$plus(s(x),y) \to s(plus(x,y))$$

$$times(0,x) \to 0$$

$$times(s(x),y) \to plus(y,times(x,y))$$

Note that this system is orthogonal. To reduce a term $plus(s,t)$, the subterm s, if not of the form 0 or $s(...)$, is strongly needed since it distinguishes between the two reductions. The same remark applies to a term $times(s,t)$. Let's consider the term

$$plus(times(0,s(0)),times(s(0),s(0)))$$

We show that the redex $times(0,s(0))$ is strongly needed. The maximal redexes are $times(0,s(0))$ and $times(s(0),s(0))$. Replacing these with $\square$ we obtain the context $plus(\square,\square)$. Replacing the first occurrence of $\square$ with p (a new term) and doing $\square$ reduction on the whole term, we obtain the term in $\square$ normal form

$$plus(p,\square)$$

No $\square$ reduction is possible. We also try this on the other occurrence of $\square$, giving the term $plus(\square,p)$. This term can be reduced to $\square$ using the $\square$ reduction $plus(0,x) \to \square$. Since p does not occur in the $\square$ normal form, the redex $times(s(0),s(0))$ is not strongly needed. In the actual system, the redex $times(s(0),s(0))$ is needed. However, for strong sequentiality we are not considering right-hand sides of rules, so we might have had the following system:

$$plus(0,x) \to a$$

$$plus(s(x),y) \to b$$

$$times(0,x) \to 0$$

$$times(s(x),y) \to plus(y,times(x,y))$$

In this system, the term t in an expression $plus(s,t)$ is not needed to compute the normal form.

Note that the conventional definition of *or* cannot be discussed in this framework since it is not orthogonal:

$$or(x, true) \rightarrow true$$

$$or(true, x) \rightarrow true$$

This is not orthogonal since the two left-hand sides unify. It is, however, "weakly orthogonal." This is a common example in which it is not possible to determine a needed redex, since in an expression $or(s,t)$ one doesn't know which of s and t will reduce to $true$. A common example of sequentiality is the $IF\ THEN\ ELSE$ construct; with usual definitions, in a term of the form $IF\ r\ THEN\ s\ ELSE\ t$, r must first be reduced to $true$ or $false$ before the rest of the expression can be reduced. Thus the redex r is needed.

We mention that Hoffman and ODonnell [HO79] define strongly left sequential systems, which permit efficient redex finding during reduction. ODonnell [O'D85] gives methods for efficient computer implementation for this class of systems; they provide a clean and powerful functional programming language. A modification of the concept of sequentiality suitable for parallel rewriting is given in [SPR89].

14 Termination orderings

We give techniques to show that a term rewriting system is terminating. These all make use of well founded partial orderings on terms having the property that if $s \rightarrow t$ then $s > t$. If such an ordering exists, then a rewriting system is terminating since infinite reduction sequences correspond to infinite descending sequences of terms in the ordering. Recall from section 9 that a termination ordering is a well-founded ordering that has the full invariance and replacement properties. We now re-state the relevant property of these orderings, given before in theorem 9.0.7:

Theorem 14.0.1. *Suppose $>$ is a termination ordering. Suppose that R is a term rewriting system and for all rules $l \rightarrow r$ in R, $l > r$. Then R is terminating.*

The simplest kind of termination orderings are those that are based on size. Let $|s|$ be the size (number of symbol occurrences) of a term s. We can then define $>$ so that $s > t$ if for all Θ making $s\Theta$ and $t\Theta$ ground terms, $|s\Theta| > |t\Theta|$. For example, $f(x,y) > g(y)$ in this ordering, but we do not have $h(x,a,b) > f(x,x)$ since x could be replaced by a large term. This termination ordering is computable; $s > t$ iff $|s| > |t|$ and no variable occurs more times in t than s. Unfortunately, this ordering is too weak to handle many interesting systems such as those containing the rule $x * (y + z) \rightarrow x * y + x * z$, since the right hand side is bigger than the left-hand side and has more occurrences of x. This ordering can be modified to weigh different symbols differently; we can modify the definition of $|s|$ for this to be a weighted sum of the number of occurrences of the symbols. This is similar to the ordering of Knuth and Bendix [KB70], but not as refined, in a couple of ways. Note that using this ordering we cannot distinguish $(x * y) * z$ from $x * (y * z)$, and so no system containing the rule $(x * y) * z \to x * (y * z)$ can be shown to terminate. However, this fact is an advantage when finding termination orderings relative to the associative and commutative axioms; for such systems, one does not want to distinguish terms that can be obtained from each other by the associative and commutative axioms. Also, the length of a reduction sequence from a term t is at most $|t|$ using this ordering, which severely restricts its applicability. The ordering of Knuth and Bendix is more refined and is able to show that systems containing the rule $(x * y) * z \to x * (y * z)$ terminate. In fact, one version of Knuth and Bendix' ordering can show termination of systems computing non-primitive recursive functions [HL89]; for such systems, the length of the rewrite sequences from a term t cannot be bounded by a primitive recursive function in the size of t.

Another class of termination orderings are the polynomial orderings suggested by Lankford [Lan75, Lan79]. For these, we interpret each function and constant symbol as a polynomial with integer coefficients. Then a term is interpreted as a function into the integers. For example, if $f(x, y)$ is interpreted as $2 * x + y$ then $f(f(x, y), y)$ is interpreted as $2 * (2 * x + y) + y$, or, $4x + 3y$. We restrict the polynomials to have non-negative values on non-negative arguments, and to be monotonic; one way to do this is to require the coefficients to be positive. Let s^I be the interpretation of a term s in this manner. Then we can then say that $s > t$ if $s^I > t^I$ for all positive values of the variables in s and t. For example, with f interpreted as above, $f(f(x, y), y) > f(x, y)$ since $4x + 3y > 2x + y$ for positive x and y. This defines a termination ordering, since the positive integers are well-ordered by $>$. The replacement property of this ordering is guaranteed by the monotonicity restriction on polynomials. This technique has the advantage of flexibility, since different polynomials can be designed for different systems; however, there are some systems for which no such polynomial ordering will work [Der87]. Also, it is not easy to devise polynomials that will suffice for termination proofs, given a term rewriting system R. For some recent results and refinements of this method, see [SZ90]. It is known that if the termination of R can be shown using such polynomials, then the length of the reduction sequences from a term t are at most double exponential [HL89] in the size of t. This limits the kind of functions that can be computed by systems whose termination can be shown with these orderings.

Before introducing some more commonly used orderings, we discuss the issue of incrementality. We say an ordering $>'$ is more powerful than (or extends) $>$ if $x > y$ implies $x >' y$, but not vice versa. It is useful to be able to define termination orderings that are partial orderings and to be able to extend them to more powerful orderings. At first sight, this may not seem reasonable, since if we can show R terminating using an ordering, we can also show R terminating using a more powerful termination ordering. However, when running the Knuth-Bendix completion method, we may not know R in advance; R is constructed incrementally as new critical pairs are generated and oriented into rules. The incremental property allows us to gradually extend the ordering to adapt to the new rules as they are generated. This is why partial orderings are of interest.

One of the most remarkable results in this area is a theorem of Dershowitz about simplification orderings, that gives a general technique for showing that an ordering is a termination ordering. Before his theorem, each ordering had to be shown well founded separately, and this was often difficult. First we define a simplification ordering.

Definition 14.0.2. A partial ordering $>$ on terms is a *simplification ordering* if it satisfies the replacement property (see section 9) and has the *subterm property*, that is, $s > t$ if t is a proper subterm of s. Also, if there are function symbols f with variable arity, we require that $f(...s...) > f(......)$ for all such f.

Theorem 14.0.3. *All simplification orderings are well founded.*

Proof. Based on Kruskal's tree theorem[Kru60], which says that in any infinite sequence $t_1 t_2 t_3 ...$ of terms, there is an i and j with $i < j$ such that t_i is embedded in t_j in a certain sense. It turns out that if t_i is embedded in t_j then $t_j \geq t_i$ for any simplification ordering $>$. ∎

Using this theorem, we can extend the polynomial orderings given above to polynomial orderings over the reals. It is no longer necessary that the range of the polynomials be well-ordered; well-foundedness is guaranteed by the theorem. For details, see [Der82]. An interesting sidelight is that it is then possible to use Tarski's decision procedure for polynomials over the reals to determine if polynomials exist that can be used to prove termination of a given system. This automates the method, but at a considerable cost since the best upper bounds for Tarski's decision procedure are still large [GV88]. It is not known what class of systems can be shown to terminate using such orderings, in terms of the length of the reduction sequences from t as a function of $|t|$. One would expect that they are more powerful than the polynomial orderings over the integers.

We now discuss the recursive path ordering [Der82] and its many variants. The power of this ordering is shown by the fact that for any primitive recursive function, there is a term rewriting system R computing it such that the termination of R can be shown using a recursive path ordering [Pla78]. In fact, [Chc90, Hof90] recently showed that if the termination of R can be shown using a recursive path ordering, then R computes a primitive recursive function (that is, the function from a term to its R-normal form is primitive recursive). Also, the recursive path orderings can easily handle the distributive rule $x * (y + z) \rightarrow x * y + x * z$. In addition, for many common systems, it is easy to find a recursive path ordering that suffices for termination proofs, in contrast to polynomial orderings. However, these orderings cannot handle the associativity rule $(x * y) * z \rightarrow x * (y * z)$; for that, "status" (defined below) needs to be added.

The recursive path ordering is defined in terms of a *precedence* ordering on function symbols. This is a partial ordering on the function symbols. We will say $f < g$ to indicate a precedence relation between symbols f and g. We will present the recursive path ordering by a complete set of inference rules that may be used to construct proofs of $s > t$. That is, if $s > t$ then there is a proof of this in the system. Also, by using the inference rules backwards in a goal-directed manner, it is possible to construct a reasonably efficient decision procedure for statements of the form $s > t$. Recall that if $>$ is an ordering, then $\gg$ is the extension of this ordering to multisets. The ordering we present is actually weaker than that usually given in the literature; we will remedy this later.

$$\frac{f = g \qquad \{s_1 ... s_m\} \gg \{t_1 ... t_n\}}{f(s_1 ... s_m) > g(t_1 ... t_n)}$$

$$\frac{s_i \geq t}{f(s_1 ... s_m) > t}$$

$$\frac{true}{s \geq s}$$

$$\frac{f > g \qquad f(s_1 ... s_m) > t_i \ all \ i}{f(s_1 ... s_m) > g(t_1 ... t_n)}$$

For example, suppose $* > +$. Then we can show that $x * (y + z) > x * y + x * z$ as follows:

$$\cfrac{\cfrac{\cfrac{\cfrac{true}{y+z > y}}{\{x, y+z\} \gg \{x, y\}}}{x * (y+z) > x * y} \quad \cfrac{\cfrac{\cfrac{true}{y+z > z}}{\{x, y+z\} \gg \{x, z\}}}{x * (y+z) > x * z} \quad * > +}{x * (y+z) > x * y + x * z}$$

An advantage of the recursive path ordering is that it is natural in many cases to find the precedence. For example, suppose a function f is defined so that $f(s)$ is defined in terms of $f(t)$ for t simpler than s (say, t are subterms of s). Thus $f(s) = H(f(t_1), ..., f(t_n))$ where H is some term involving the $f(t_i)$. Suppose H can be expressed as a composition of functions that have previously been defined. Then we can choose the precedence so that if g has been previously defined, then $f > g$. Then $f(s) > H(f(t_1), ..., f(t_n))$ using the recursive path ordering, since $s > t_i$ for all i, implying that $f(s) > f(t_i)$ for all i. Also, $f(s) > g(s_1, ..., s_n)$ if $f(s) > s_i$ for all i. Using this repeatedly, we obtain that $f(s) > H(f(t_1), ..., f(t_n))$. Thus we can prove termination of sets of rules including the rule $f(s) \rightarrow H(f(t_1), ..., f(t_n))$.

We note that the recursive path ordering has the *incrementality* property in terms of the precedence. That is, let $<_{rpo(<)}$ be the recursive path ordering for a precedence $<$. Then if $<'$ extends $<$, $<_{rpo(<')}$ extends $<_{rpo(<)}$. This is convenient when running Knuth-Bendix completion: when orienting a new equation, we can extend the precedence as necessary, knowing that all rules previously oriented will still be oriented the same way.

We show some of the features of this ordering by looking at a rewrite system to compute the factorial function:

$$fact(s(x)) \rightarrow times(s(x), fact(x))$$

$$fact(0) \rightarrow s(0)$$

$$times(0, x) \rightarrow 0$$

$$times(s(x), y) \rightarrow plus(y, times(x, y))$$

$$plus(0, x) \rightarrow x$$

$$plus(s(x), y) \rightarrow s(plus(x, y))$$

For this system, we choose a precedence with $fact > times > plus > s > 0$. This seems natural, since "$fact$" is defined in terms of "$times$", "$times$" is defined in terms of "$plus$", and "$plus$" is defined in terms of "s" and "0". Using this precedence and the recursive path ordering, we have $l > r$ for all rules $l \rightarrow r$ in the above system. The reason is that the dominant term in the ordering is the one with the largest function symbol. For the first rule $fact(s(x)) \rightarrow times(s(x), fact(x))$, the dominant term on the left is $fact(s(x))$ and the dominant term on the right is $fact(x)$. Since $s(x) > x$, $fact(s(x)) > fact(x)$ so the left-hand side is larger in the ordering, even thought the right-hand side has larger size. This gives a proof that the computation of factorial using this system, terminates. Sometimes there is not a single dominant term, in which case the definition of the recursive path ordering has to be used more carefully. It should be clear that the same ideas will serve to find a precedence in which any function defined primitive recursively, can be shown to terminate.

In order to handle the associativity rule $(x * y) * z \rightarrow x * (y * z)$ it is necessary to modify the ordering so that subterms are considered lexicographically. In general, if $>$ is an ordering, then the lexicographic extension $>_{lex}$ of $>$ to fixed-length tuples is defined as follows:

$$\frac{s_1 > t_1}{(s_1 \ldots s_m) >_{lex} (t_1 \ldots t_n)}$$

$$\frac{s_1 = t_1 \qquad (s_2 \ldots s_m) >_{lex} (t_2 \ldots t_n)}{(s_1 \ldots s_m) >_{lex} (t_1 \ldots t_n)}$$

Sometimes the ordering is defined on tuples of different lengths; for this we also need the rule

$$\frac{true}{(s_1 \ldots s_m) >_{lex} ()}$$

One can show that if $>$ is well founded, then so is its extension $>_{lex}$. This lexicographic treatment of subterms is the idea of the lexicographic path ordering of Kamin and Levy. This ordering is defined by the following inference rules:

$$\frac{f = g \qquad (s_1 \ldots s_m) >_{lex} (t_1 \ldots t_n) \qquad f(s_1 \ldots s_m) > t_j,\ all\ j \geq 2}{f(s_1 \ldots s_m) > g(t_1 \ldots t_n)}$$

$$\frac{s_i \geq t}{f(s_1 \ldots s_m) > t}$$

$$\frac{true}{s \geq s}$$

$$\frac{f > g \qquad f(s_1 \ldots s_m) > t_i\ all\ i}{f(s_1 \ldots s_m) > g(t_1 \ldots t_n)}$$

In the first inference rule, we don't need to test $s > t_1$ since $(s_1 \ldots s_m) >_{lex} (t_1 \ldots t_n)$ implies $s_1 \geq t_1$ hence $s > t_1$. One can show that this is a simplification ordering for systems having fixed arity function symbols. It is interesting that in this system, one can prove the termination of Ackermann's function, so the restriction to primitive recursive functions does not apply. There are also many orderings intermediate between the recursive path ordering and the lexicographic path ordering; these are known as orderings with "status." The idea of status is that for some function symbols, when $f(s_1 \ldots s_m)$ and $f(t_1 \ldots t_n)$ are compared, the subterms are compared using the multiset ordering. For other function symbols, the subterms are compared using the lexicographic ordering. For other function symbols, the subterms are compared using the lexicographic ordering in reverse, that is, from right to left; this is equivalent to reversing the lists and then applying thee lexicographic ordering. One can show that all such versions of the orderings are simplification orderings, for function symbols of bounded arity.

There are also many other orderings known that are similar to the above ones, such as the recursive decomposition ordering [JLR82] and others; for some surveys see [Der87, Ste89]. In practice, ***quasi-orderings*** are often used to prove termination. A relation is a quasi-ordering if it is reflexive and transitive. A quasi-ordering is often written as $\geq$. Thus we have that $x \geq x$ for all x, and if $x \geq y$ and $y \geq z$ then $x \geq z$. It is possible that $x \geq y$ and $y \geq x$ even if x and y are distinct; such x and y are in some sense "equivalent" in the ordering. We write $x > y$ if $x \geq y$ but not $y \geq x$, for a quasi-ordering $\geq$. The relation $>$ is called the *strict part* of the quasi-ordering $\geq$. Note that the strict part of a quasi-ordering is a partial ordering. The multiset extension of a quasi-ordering is defined in a manner similar to the multiset extension of a partial ordering [JL82, Der87]. We write $s \approx t$ if s and t are identical up to a permutation of arguments.

Definition 14.0.4. A quasi-ordering $\geq$ on terms satisfies the *replacement property* (is *monotonic*) if $s \geq t$ implies $f(...s...) \geq f(...t...)$. Note that it is possible to have $s > t$ and $f(...s...) \approx f(...\mathrm{t}...)$. (See above for a definition of $\approx$.)

Definition 14.0.5. A quasi-ordering $\geq$ is a *quasi-simplification ordering* if $f(...t...) \geq t$ for all terms and if $f(...t...) \geq f(......)$ for all terms and all function symbols f of variable arity, and if the ordering satisfies the replacement property.

Definition 14.0.6. A quasi-ordering $\geq$ satisfies the *full invariance property* (see section 9) if $s > t$ implies $s\Theta > t\Theta$ for all s, t, Θ.

Theorem 14.0.7 (Dershowitz [Der82]). *For terms over a finite set of function symbols, all quasi-simplification orderings have strict parts which are well founded.*

Proof. Using Kruskal's tree theorem [Kru60]. ∎

Theorem 14.0.8. *Suppose R is a term rewriting system and $\geq$ is a quasi-simplification ordering which is satisfies the full invariance property. Suppose that for all rules $l \rightarrow r$ in R, $l > r$. Then R is terminating.*

Proof. Suppose $s_1 \rightarrow s_2 \rightarrow ...s_n \rightarrow ....$ Then $s_1 \geq s_2 \geq s_3...$ since the ordering is monotonic. We only know that the strict part of $\geq$ is well founded, so this does not help us. However, let $M(s)$ be the multiset of subterms of s. One can show that $M(s_1) \gg M(s_2) \gg M(s_3)...$ where $\ggeq$ is the multiset extension of $\geq$ and $\gg$ is its strict part. Since the strict part of $\geq$ is well founded for terms over a finite set of function symbols, it turns out that $\gg$ is well founded also. Therefore this sequence must be finite. Thus R terminates. Note that since R is finite, only finitely many function symbols will appear in the sequence $s_1 \rightarrow s_2 \rightarrow ...s_n \rightarrow ....$ ∎

Sometimes it is convenient to define a quasi-ordering to show termination. For example, we can define the (quasi) recursive path ordering as a quasi-simplification ordering using the following inference rules:

$$\frac{f = g \qquad \{s_1...s_m\} \underline{\gg} \{t_1...t_n\}}{f(s_1...s_m) \geq g(t_1...t_n)}$$

$$\frac{f = g \qquad \{s_1...s_m\} \gg \{t_1...t_n\}}{f(s_1...s_m) > g(t_1...t_n)}$$

$$\frac{true}{s \geq s}$$

$$\frac{s_i \geq t}{f(s_1...s_m) > t}$$

$$\frac{f > g \qquad f(s_1...s_m) > t_i \; all \; i}{f(s_1...s_m) > g(t_1...t_n)}$$

Here $\underline{\gg}$ is the extension of $\geq$ to multisets. This is defined in section 9. Note that $s > t$ if $s \geq t$ but not $t \geq s$.

Actually, this is known as the recursive path ordering in the literature. The idea is that terms that are identical up to permutation of arguments, are equivalent. For example, we want to show that $f(g(a,b),d) > f(g(b,a),c)$ if $d > c$. Using the above inference rules, we have that $g(a,b) \approx g(b,a)$ hence $\{g(a,b),d\} \underline{\gg} \{g(b,a),c\}$ so $f(g(a,b),d) > f(g(b,a),c)$. However, using the previous inference rules for the recursive path ordering, these two terms cannot be compared since the multisets $\{g(a,b),d\}$ and $\{g(b,a),c\}$ cannot be compared. Another way to obtain this ordering is to add inference rules as follows to the previous ones.

$$\frac{s > t \qquad t \approx u}{s > u}$$

$$\frac{t > s \qquad t \approx u}{u > s}$$

For quasi-orderings, the precedence on function symbols can also be a quasi-ordering; this means essentially that different functions can be grouped into equivalence classes. There are a number of different orderings like the recursive path ordering. However, it turns out that they all agree when the precedence is total [Ste89, Les90], if the status is assigned the same way. Further, they all have the incrementality property. This suggests a method of defining and computing a "maximal" termination ordering for this class. Let $\geq_{rpo(\geq)}$ be the recursive path (quasi-)ordering (with status) for a given precedence $\geq$. Define $\geq_{rpo'(\geq)}$ so that $s \geq_{rpo'(\geq)} t$ if $s \geq_{rpo(\geq')} t$ for all extensions $\geq'$ of $\geq$ to a total ordering. This ordering is then as powerful as all the other orderings including the recursive path ordering, the path of subterms ordering [Pla78], the recursive decomposition ordering [JLR82], and others. Computing $\geq_{rpo'}$ directly in this way may be expensive, since there may be many extensions of $\geq$ to a total precedence $\geq'$. A possible optimization is to use the definition of the recursive path ordering to construct a Boolean expression $B(\geq)$ giving conditions on $\geq'$ under which $s \geq' t$. Then we only have to test whether all extensions of $\geq$ satisfy B.

For example, suppose we are comparing $f(a)$ and $g(b)$ in the recursive path ordering, and the precedence on function symbols is that $f > b$ and $a > g$. We use the inference rules backwards, working from the goal of $f(a) \geq g(b)$ or $g(b) \geq f(a)$. First we try to show $f(a) \geq g(b)$. The first and third inference rules for the recursive path ordering don't apply, since we don't have an ordering between f and g. The only one that can work is the second one, which becomes

$$\frac{a \geq g(b)}{f(a) > g(b)}$$

We know that $a > g$, so we have the inference rule

$$\frac{a \geq b}{a > g(b)}$$

However, the relation between a and b is not specified, so the proof cannot be completed. Suppose we try to prove $g(b) \geq f(a)$. This also fails. As before, the only rule we can apply is

$$\frac{b \geq f(a)}{g(b) > f(a)}$$

Now, none of the rules can be used to show that $b \geq f(a)$. But we can show that in any extension of the precedence to a total ordering $\geq'$ on function symbols, $f(a) >' g(b)$ where $>'$ is the strict part of the recursive path ordering for the precedence $\geq'$. If $f \geq a$ then $f > g$ so f is the maximal symbol of the two terms and $f(a) >' g(b)$. If $a \geq f$ then a is the maximal symbol of the two terms so $f(a) >' g(b)$. One of these must hold in a total extension of this precedence on function symbols. Therefore $f(a) >_{rpo'(\geq)} g(b)$.

One way to test if $s \geq_{rpo(\geq)} t$, as mentioned above, is to construct a Boolean expression involving $\geq$ and function symbols such that $s \geq t$ iff $B(\geq)$ is true. This can be derived systematically from the rules for the recursive path ordering. To decide whether $s \geq_{rpo'(\geq)} t$, we need to show that all extensions of $\geq$ to a total precedence $\geq'$ satisfy $B(\geq')$. This avoids repeated computations on the term structures. The precedence $\geq$ can itself be defined by a conjunction of inequalities of the form $f_1 > g_1$, $f_2 > g_2$, et cetera, with possibly some statements like $f_j \geq g_j$. Let's call this conjunction of inequalities on function symbols $C(\geq)$. Let $T(\geq)$ be the axioms of total quasi-orderings, namely, symmetry, transitivity, and $x \geq y \vee y \geq x$ for all x,y. We can represent $x > y$ as $(x \geq y \wedge (\neg y \geq x))$. Then we need to show that $C(\geq) \wedge T(\geq) \supset B(\geq)$. This is a simple theorem proving problem, and can be approached by a number of methods that are reasonably efficient on small formulas. It is possible that this ordering can be made even stronger, by looking more carefully at variables. For example, it is conceivable that we might not have $s > t$ even though for all substitutions Θ making s and t ground terms, $s\Theta > t\Theta$. So, we could define another extension $>'$ of an ordering $>$ by $s >' t$ if for all Θ making $s\Theta$ and $t\Theta$ ground, $s\Theta > t\Theta$. Combining this idea with the previous idea of maximal extension orderings might yield a more powerful ordering. However, it is possible that this ordering based on Θ is not even decidable, or difficult to decide. Jouannaud and Okada [JO91] show that the existential fragment of the theory of the recursive path ordering is decidable, which may help here.

Some decidability results about termination are known. In general, it is undecidable whether a system R is terminating [HL78]; however, for ground systems, that is, systems in which left and right-hand sides of rules are ground terms, termination is decidable [HL78]. For non-ground systems, termination of even one rule systems has recently been shown to be undecidable [Dau89]. The concept of *fair termination* was defined in [Tis89]; the idea is that we only consider reduction sequences in which a redex that persists is eventually contracted. (That is, no redex persists indefinitely and is never contracted.) It was shown in [Tis89] that fair termination is decidable for ground systems.

There are also some results about termination for orthogonal systems; for example, it is known that a system is weakly innermost normalizing (every term has a normal form obtainable by some innermost rewriting sequence) iff it is terminating. Also, for systems that are *non-erasing* (the left and right-hand sides of rules have the same variables appearing), a system is weakly normalizing (every term has a normal form) iff it is terminating.

A number of relationships between termination orderings and large ordinals have been found; this is only natural since any well-founded ordering corresponds to some ordinal. It is interesting that the recursive path ordering and other orderings provide intuitive and useful descriptions of large ordinals. For a discussion of this, see [Der87] and [Der80].

There has also been some work on modular properties of termination; for example, if we know that R_1 and R_2 terminate, what can we say about $R_1 \cup R_2$ under certain conditions? For a few examples of works along this line, see [Toy87, TKB89, Mid90].

15 Extensions to rewriting

We now briefly discuss some extensions to rewriting, namely, equational rewriting, rewriting infinite terms, and conditional rewriting. For more detailed accounts, see [DJ90, DKP91, DOS88].

15.1 Equational rewriting

There are two motivations for equational rewriting. The first is that some rules are nonterminating and cannot be used with a conventional term rewriting system. One example is the commutative axiom $x + y = y + x$ which is nonterminating no matter how it is oriented into a rewrite rule. The second reason is that if an operator like $+$ is associative and commutative then there are many equivalent ways to represent terms like $a + b + c + d$. This imposes a burden in storage and time on a theorem prover or term rewriting system. Equational rewriting permits us to treat some axioms, like $x + y = y + x$, in a special way, avoiding problems with termination. It also permits us to avoid explicitly representing many equivalent forms of a term. The cost is a more complicated rewriting relation, more difficult termination proofs, and a more complicated completion procedure. Indeed, significant developments are still occurring in these areas, to attempt to deal with the problems involved.

The general idea is to consider E-equivalence classes of terms instead of single terms. The E-equivalence classes consist of terms that are provably equal under E. For example, if E includes associative and commutative axioms for $+$, then the terms $(a + b) + c$, $a + (b + c)$, $c + (b + a)$, et cetera will all be in the same E-equivalence class. Recall that $s =_E t$ if $E \models s = t$, that is, t can be obtained from s by replacing subterms using E. Note that $=_E$ is an equivalence relation. Usually some representation of the whole equivalence class is used; thus it is not necessary to store all the different terms in the class. This is a considerable savings in storage and time for term rewriting and theorem proving systems.

We need to define a rewriting relation on E-equivalence classes of terms. If s is a term, let $[s]_E$ be its E-equivalence class, that is, the set of terms E-equivalent to s. The simplest approach is to say that $[s]_E \rightarrow [t]_E$ if $s \rightarrow t$. Retracting this back to individual terms, we say that $u \rightarrow_{R/E} v$ if there are terms s and t such that $u =_E s$ and $v =_E t$ and $s \rightarrow_R t$. This system R/E is called a *class rewriting system*. However, R/E rewriting turns out to be difficult to compute, since it requires searching through all terms E-equivalant to u. A computationally simpler idea is to say that $u \rightarrow v$ if u has a subterm s such that $s =_E s'$ and $s' \rightarrow t$ and v is u with s replaced by t. In this case we say that $u \rightarrow_{R,E} v$. This system R, E is called the *extended rewrite system* for R modulo E. Note that rules with E-equivalent left-hand sides need not be kept. The R, E rewrite relation only requires using the equational theory on the chosen redex s instead of the whole term, to match s with the left-hand side of some rule. Such E-matching is often (but not always, see [HH87]) easy enough computationally to make R, E rewriting much more efficient than R/E rewriting. Unfortunately, $\rightarrow_{R/E}$ has better logical properties for deciding $R \cup E$ equivalence. So the theory of equational rewriting is largely concerned with finding connections between these two rewriting relations.

Consider the systems R/E and R, E where R is $\{a * b \rightarrow d\}$ and E consists of the associative and commutative axioms for $*$. Suppose s is $(a * c) * b$ and t is $c * d$. Then $s \rightarrow_{R/E} t$ since s is E-equivalent to $c * (a * b)$. However, it is not true that $s \rightarrow_{R,E} t$ since there is no subterm of s that is E-equivalent to $a * b$. Suppose s is $(b * a) * c$. Then $s \rightarrow_{R,E} d * c$ since $b * a$ is E-equivalent to $a * b$.

Note that if E equivalence classes are nontrivial then it's impossible for class rewriting to be confluent in the traditional sense (since any term E-equivalent to a normal form will also be a normal form of a term). So we have to modify the definition to allow E-equivalent normal forms. We want to capture the property that class rewriting is confluent when considered as a rewrite relation on equivalence classes. More precisely, R/E is *(class) confluent* if for any term t, if $t \rightarrow^*_{R/E} u$ and $t \rightarrow^*_{R/E} v$ then there are E-equivalent terms u' and v' such that $u \rightarrow^*_{R/E} u'$ and $v \rightarrow^*_{R/E} v'$. If R/E is class confluent and terminating then a term may have more than one normal form, but all of them will be E-equivalent. Furthermore, if R/E is class confluent and terminating, then we can reduce any $R \cup E$ equivalent terms to E equivalent terms by rewriting. Then an E-equivalence procedure can be used, if there is one. Note that E-equivalent rules need not both be kept, for this method.

We say that R is *Church-Rosser modulo E* if any two $R^= \cup E$-equivalent terms can be R, E rewritten to E-equivalent terms. This is not the same as saying that R/E is Church-Rosser, considered as a rewrite system on E-equivalence classes. Note that R, E rewriting is a subset of R/E rewriting, so if R/E is terminating, so is R, E. If R/E is terminating and R is Church-Rosser modulo E then R, E rewriting is also terminating and $R^= \cup E$-equality is decidable if E-equality is. Also, we can use the computationally simpler R, E rewriting to decide the equational theory. But Church-Rosser modulo E is not a local property; as stated, it's not obvious that the property is decidable. Therefore we desire decidable sufficient conditions for it. This is the contribution of

Jouannaud and Kirchner [JK86], using confluence and "coherence". The idea of coherence is that there should be some similarity in the way all elements of an E-equivalence class rewrite. Their conditions involve critical pairs between rules and equations and E-unification procedures.

Another approach is to add new rules to R to obtain a logically equivalent system R'/E; that is, $R^= \cup E$ and $R'^= \cup E$ have the same logical consequences (i.e., they are equivalent), but R', E rewriting is the same as R/E rewriting. Therefore we can use the computationally simpler R', E rewriting to decide the equality theory of R/E. This is done for associative-commutative operators by Peterson and Stickel [PS81]. In this case, confluence can be decided by methods simpler than those of Jouannaud and Kirchner Termination for equational rewriting systems is tricky to decide; we will comment on that later. Another topic is completion for equational rewriting, adding rules to convert an equational rewriting system into a logically equivalent equational rewriting system with desired confluence properties. This is discussed by Peterson and Stickel [PS81] and also by Jouannaud and Kirchner [JK86]; for earlier work along this line see [LB77, LBB84].

15.2 AC rewriting

We now consider the special case of rewriting relative to the associative and commutative axioms $E = \{f(x,y) = f(y,x), f(f(x,y),z) = f(x,f(y,z))\}$ for a function symbol f. Special efficient methods exist for this case. One idea is to modify the term structure so that R, E rewriting can be used rather than R/E rewriting. This is done by of *flattening*, that is, a term $f(s_1, f(s_2, ..., f(s_{n-1}, s_n)..))$, where none of the s_i have f as a top-level function symbol, is represented as $f(s_1, s_2, ..., s_n)$. Here f is a vary-adic symbol, which can take a variable number of arguments. Similarly, $f(f(s_1, s_2), s_3)$ is represented as $f(s_1, s_2, s_3)$. This represents all terms that are equivalent up to the associative equation $f(f(x,y),z) = f(x,f(y,z))$ by the same term. Also, terms that are equivalent up to permutation of arguments of f are also considered as identical. This means that each E-equivalence class is represented by a single term. This also means that all members of a given E-equivalence class have the same term structure, making R, E rewriting seem more of a possibility. Note however that the subterm structure has been changed; $f(s_1, s_2)$ is a subterm of $f(f(s_1, s_2), s_3)$ but there is no corresponding subterm of $f(s_1, s_2, s_3)$. This means that R, E rewriting does not simulate R/E rewriting on the original system. For example, consider the systems R/E and R, E where R is $\{a * b \rightarrow d\}$ and E consists of the associative and commutative axioms for $*$. Suppose s is $(a * b) * c$ and t is $d * c$. Then $s \rightarrow_{R/E} t$; in fact, $s \rightarrow_{R,E} t$. However, if we flatten the terms, then s becomes $*(a, b, c)$ and s no longer rewrites to t since the subterm $a * b$ has disappeared.

To overcome this, we add *extensions* to rewrite rules to simulate their effect on flattened terms. The extension of the rule $\{a * b \rightarrow d\}$ is $\{*(x, a, b) \rightarrow *(x, d)\}$, where x is a new variable. With this extended rule, we do have that $*(a, b, c)$ rewrites to $d * c$. The general idea, then, is to flatten terms, and extend R by adding extensions of rewrite rules to it. Then, extended rewriting on flattened terms using the extended R is equivalent to class rewriting on the original R. Formally, suppose s and t are terms and s' and t' are their flattened forms. Suppose R is a term rewriting system and R' is R with the extensions added. Suppose E is associativity and commutativity. Then $s \rightarrow_{R/E} t$ iff $s' \rightarrow_{R',E} t'$. The extended R is obtained by adding, for each rule of the form $f(r_1, r_2, ..., r_n) \rightarrow s$

where f is associative and commutative, an extended rule of the form $f(x, r_1, r_2, ..., r_n) \rightarrow f(x, s)$, where x is a new variable. The original rule is also retained. This idea does not always work on other equational theories, however. Note that some kind of associative-commutative matching is needed for extended rewriting. This can be fairly expensive, since there are so many permutations to consider, but it is fairly straightforward to implement. Completion relative to associativity and commutativity can be done with the flattened representation; a method for this is given in [PS81]. This method requires associative-commutative unification.

An interesting idea from [Pla84] is to use the flattened term representation for associative-commutative operators but change the representation slightly, to a multiset notation. We note that the arguments of an associative-commutative function are really a multiset, since order does not matter, but the number of occurrences matters. Therefore, we can represent $f(s, s, t, t, t)$, for example, by $f(2 \circ s, 3 \circ t)$. The advantage of this is that terms with many occurrences of the same argument can be represented much more compactly. This permits us, for example, to represent n as $1 + 1 + ... + 1$ (n times), where $+$ is associative and commutative, and perform arithmetic by associative-commutative term rewriting without much loss of efficiency. For example, in this representation, unifying $3 + x$ with 5 produces the result $x = 2$. Unifying $2x + 3y$ with 15 gives the answers $x = 3, y = 3$ and $x = 6, y = 1$. If we also use the identity in the unification algorithm, we can get the answer $x = 0, y = 5$. We can also express $x \leq y$ by $x + z = y$. It would be interesting to see what is the effect of this representation on the complexity of associative-commutative matching and unification. (For some complexity results for the usual case, see [BKN85, KN86].)

This brings up the issue of how to perform arithmetic efficiently in term rewriting systems. The axiomatic approach is to represent integers in successor notation, as in the Peano axioms, such as $s(s(s(0)))$ for 3. This is too inefficient. The multiset notation helps some, but what about division and other special functions? Possibly we can have infinitely many rewrite rules to evaluate arithmetic expressions, implemented by some kind of an arithmetic procedure, but this might raise theoretical problems for testing termination and confluence and performing completion.

One approach to associative-commutative theories used by the Argonne group is to associate to the right and sort the arguments to an associative-commutative function in some canonical way. Although this method does not have the power of general associative-commutative matching and unification, it is simple to implement and often has good performance. One can show that it is essentially equivalent to a version of unfailing completion on ground terms.

Another recent development of interest is a new method of AC unification due to Domenjoud [Dom91]. The idea is to add a new identity, obtaining a kind of associative-commutative system with identity, which turns out to have a much simpler unification algorithm. This requires a careful treatment of the new symbol, however, for correctness. Still, it will be interesting to see if this method dramatically increases the power of associative-commutative term rewriting systems and theorem provers.

15.3 Left-commutativity

We now give another approach to associative-commutative theories which uses a simpler unification algorithm. This approach provides a unification algorithm with more predictable behavior than AC unification, and so it may be more suitable for implementation of functional programming languages with sets or multisets and narrowing. The idea is to have a list constructor *cons* and represent lists as in Lisp or Prolog, so that $[x, y, z]$ is $cons(x,\ cons(y,\ cons(z,\ [])))$ where $[]$ is a constant symbol for the end of a list. We also abbreviate $cons(x, y)$ by $[x|y]$. Then instead of flattening an associative-commutative operator f, we represent $f(x_1, ..., x_n)$ by $f([x_1, ..., x_n])$, abbreviated by $f[x_1, ..., x_n]$. Thus, f becomes a unary function symbol that takes a list as an argument. We also have the left-commutativity property for "cons," namely, $cons(x, cons(y, z)) = cons(y, cons(x, z))$. This means that the order of elements in a list does not matter, just the number of occurrences. Thus we really have multisets. This reflects the associativity and commutativity of f. Note that only one multiset constructor "cons" is needed even if there is more than one associative-commutative operator. To preserve the equational theory, we also need the *flattening equation* $f[f[x|y]|z] = f[f(y)|[x|z]]$. If f does not have an identity then this has to be represented as $f[f[x|[y|w]]|z] = f[f[y|w]|[x|z]]$ to prevent the formation of the term $f([])$ which would be an identity for f. These equations, oriented as rules from left to right, cause flattening by term rewriting. We also need the equation $f([x]) = x$. Finally, $f(x, y)$ is replaced by $f[x, y]$ everywhere, that is, $f([x, y])$. Then, we can show that the equational theory is preserved. More precisely, let R, E be the original system, where E is associativity and commutativity of some operators, and let R', E' be the modified system. Here E' contains only the left-commutativity of list construction; the other new equations are rewrite rules, oriented from left to right. Suppose r and s are terms and r' and s' are their representations using lists in this way. Then $R, E \models r = s$ iff $R', E' \models r' = s'$.

For this theory E', E'-unification is simple. We sketch a unification algorithm, leaving out obvious details that parallel the usual unification algorithm in the absence of an equational theory. To unify $f(x_1, ..., x_n)$ with $f(y_1, ..., y_n)$ where f is any symbol other than the left-commutative list constructor, we recursively unify x_i and y_i as in the standard unification algorithm. To unify lists, we distinguish whether the lists end with a $[]$ or with a *tail variable* such as y in the list $cons(a, cons(b, cons(c, y)))$, represented by convention as $[a, b, c|y]$. Note that a tail variable appears in the flattening rule above. Also, we will introduce more tail variables below. It turns out that tail variables do not appear in any other position in terms in our systems except as tail variables, possibly in two places (on the left-hand side and on the right-hand side of an equation). This simplifies the unification. Suppose we are unifying two lists, $[s_1, s_2, ..., s_m]$ and $[t_1, t_2, ..., t_n|y]$ where y is a tail variable. If $n > m$ the unification fails. Otherwise, we replace y by a list of $m - n$ new variables $[z_1, ..., z_{m-n}]$ and pick an arbitrary permutation of $[t_1, t_2, ..., t_n, z_1, ..., z_{m-n}]$ and unify corresponding arguments in both lists, recursively. Note that this may yield more than one most general unifier. If we are unifying $[s_1, s_2, ..., s_m|x]$ and $[t_1, t_2, ..., t_n|y]$, where x and y are tail variables, we pick an integer k, $0 \leq k, k \leq m, k \leq n$, and pick a subset of k of the list elements from both lists. We pick a one-to-one correspondence between these elements and unify each element of one list with the corresponding element of the other list, recursively. Then we have $m + n - 2k$ elements left over. These are inserted into the most general instance list as is, with a new tail variable. More precisely,

x is bound to a list of the left-over elements from the second list, followed by a new tail variable z, and y is bound to a list of the left-over elements from the first list, followed by the tail variable z.

A similar, but more complicated unification algorithm for this theory is given in [Jea80, Kir85]. Our algorithm is simpler because the lists are restricted to be well-formed, that is, to end with [] or a tail variable. Also, it turns out that R/E' rewriting and R, E' rewriting coincide (up to E' equivalence). Furthermore, the critical pair test can be modified for this system simply by replacing ordinary unification by E'-unification. Therefore completion and rewriting can readily be adapted to this framework. This representation permits the deletion of terms that are identical up to permutations of lists

A problem with this approach is that finite confluent systems often do not exist. For example, if the rule $f(x, g(x)) \rightarrow e$ is in the original system, then for confluence in the modified system we need the infinitely many rules of the form $f[x, y_1, y_2, ..., y_n, g(x)] \rightarrow f[y_1, y_2, ..., y_n]$. These rules will be generated automatically by completion. This system is suitable for theorem proving purposes, however. A way to eliminate the need for all these rules is to add extensions to each rewrite rule, involving new tail variables. For example, the rule $f(x, g(x)) \rightarrow e$ is represented as $f[x, g(x)] \rightarrow e$. The extension in this case would be $f[x, g(x)|y] \rightarrow [e|y]$. This allows finite confluent systems more often. However, sometimes it is necessary to have more than one extension. If f and g are both associative-commutative in the original system R then for a rule $f(s_1, ...s_m) \rightarrow g(t_1, ...t_n)$ we need in addition to $f[s_1, ...s_m] \rightarrow g[t_1, ...t_n]$ the extensions $f[s_1, ...s_m|z] \leftrightarrow f[g[t_1, ...t_n]|z]$ and $g[f[s_1, ...s_m]|z] \leftrightarrow g[t_1, ...t_n|z]$. The reason for the $\leftrightarrow$ is that the extensions may have a different orientation than the original rule.

Termination is straightforward in this method. We can define the *semi-flattened form* of a term as the term with $f[s_1, ..., s_n]$ replaced by $f(s_1, ..., s_n)$ everywhere. For this, a tail variable is represented as an arbitrary number of variable arguments of f. Then, the usual recursive path ordering on the semi-flattened forms is an E'-termination ordering. In fact, it's an appealing idea to just work with the semi-flattened form always, since this simplifies the term structure. However, this transforms the flattening rule into infinitely many rewrite rules, complicating the system. Also, this transforms left-commutativity into an infinite set of equations for permuting the arguments of f, although these equations may not need to be kept explicitly.

15.4 Other sets of equations

The general topic of completion for other equational theories was addressed by Jouannaud and Kirchner in [JK86]. Earlier work along these lines was done by Lankford, as mentioned above. Such completion procedures may use E-unification. Also, they may distinguish rules with linear left-hand sides from other rules. For a recent approach to associative-commutative-idempotent systems, see [BPW89]. In this approach, conditions are added to rewrite rules. There have also been some approaches based on unfailing completion. For example, Peterson [Pet90] adds conditions to rewrite rules involving inequalities between variables and terms and in this way obtains complete systems without special E-unification procedures. He might have the rule $x * y \rightarrow y * x$ with the condition

$x > y$, meaning that this rule can be applied to a term if x is bound to a value larger than y in the termination ordering being used. Another approach to equations is presented in [MN90], where some confluent and terminating systems for familiar theories are given but constraints are not explicitly kept.

It looks like there is more to be done in the way of general methods not based on any special E-unification procedure. Consider a system with an associative-commutative operator $*$. In such a system, many equations like $x * y * z = y * z * x$ are generated, in which the left and right-hand sides are permutations of one another. The same phenomenon occurs also for other identities, such as $x * g(x) = e$, which can appear in many forms such as $g(x) * z * x = z$. One way to eliminate these duplicates is by AC unification. Another way is to observe that we are really only interested in ground instances of equations, for theorem proving applications. However, no ground instance of $x * y * z = y * z * x$ is irreducible. Depending on the relative ordering of the ground terms replacing x, y, and z, both sides of the equation will reduce to the same term using associativity and commutativity and the equation $x * y * z = y * x * z$. For example, consider the term $r * s * t$ where $r > s > t$ and r, s, and t are in normal form. This rewrites to $s * r * t$ using the equation $x * y * z = y * x * z$ using the lexicographic path ordering. Further rewrites yield $t * s * r$ as the normal form. So, no ground instances of the equation $x * y * z = y * z * x$ need be retained; instead, only the normal forms corresponding to the possible orderings of x, y, and z need be kept. This is a fully general approach, not limited to associative-commutative systems, and eliminates a large number of such permutations. Also, we have to consider the case when some of the terms r, s, and t are identical. In this case, the identical terms can be interchanged without affecting the ground instance. So, if r, s, and t are identical, then we don't need the ground instance $r * s * t = s * t * r$ even though it is irreducible; the reason is that we can interchange r, s, and t to obtain the identical instance $r * s * t = r * s * t$ which is an instance of $x = x$. Using this idea, we only need to consider the cases corresponding to strict inequalities between variables and terms; the case of equalities does not need to be considered. This approach seems to be an attractive alternative to specialized equational unification, because of its simplicity and generality.

Another property of this method is illustrated by the following equation: $f(x_1, x_2, ..., x_n) = x_1 * x_2 * ... * x_n$ where $*$ is associative and commutative as above. It seems that for this equation we will need all permutations of the x_i since none of them are equivalent. However, this is not true. By completion, we can generate identities involving f, of the form $f(x_1, x_2, ..., x_n) = f(y_1, y_2, ..., y_n)$ where the y_i are a permutation of the x_i. Such equations also permit the arguments of f to be rewritten to a canonical order, reducing the number of equations that need to be kept.

15.5 Semi-Thue systems

Semi-Thue systems, or string-rewriting systems, originated historically in an attempt to investigate computability. These can be simulated by term-rewriting systems in which every function symbol is monadic (has arity one). Note that such systems have no ground terms, so every term has a variable at the end, for example, $f(g(f(f(g(x)))))$, written as $fgffg$. For such systems, rewriting corresponds to substring replacement. Another way of viewing such systems is as presentations of monoids, where there are a finite number of individual constants and an associative concatenation operator. For such systems, confluence, termination, and complexity questions similar to those for general rewriting systems, arise. There are also some connections with context-free languages. For a discussion of such systems, see the survey [Boo87] by Book.

15.6 AC termination orderings

We now consider termination orderings for special equational theories E. The problem is that E-equivalent terms are identified when doing equational rewriting, so that all E-equivalent terms have to be considered the same by the ordering. Although left-commutativity is easy to handle as shown above, equational rewriting causes considerable problems for the recursive path ordering and similar orderings. For example, consider the associative-commutative equations E. We can represent E-equivalence classes by flattened terms, as mentioned above. However, applying the recursive path ordering to such terms violates monotonicity. Suppose $* > +$ and $*$ is associative-commutative. Then $x * (y + z) > x * y + x * z$. By monotonicity, we should have $u * x * (y + z) > u * (x * y + x * z)$. In fact, this fails; the term on the right is larger in the recursive path ordering. A number of attempts have been made to overcome this. One of the earliest is the associative path ordering of Dershowitz, Hsiang, Josephson, and Plaisted [DHJP83]. This ordering applied to transformed terms, in which big operators like $*$ were pushed inside small operators like $+$. The ordering was not extended to non-ground terms in that paper, but it seems that it would be fairly simple to do so using the fact that a variable is smaller than any term properly containing it. A simpler approach was given by Bachmair and Plaisted [BP85], but this requires certain conditions on the precedence. This work was generalized by Bachmair and Dershowitz [BD86] using the idea of "commutation" between two term rewriting systems. Recently, Kapur [KSZ90] has devised a fully general associative termination ordering that applies to non-ground terms, but may be hard to compute. Another issue is the incorporation of status in such orderings, such as left-to-right, right-to-left, or multiset, for various function symbols. E-termination orderings for other equational theories may be even more complicated than for associativity and commutativity.

15.7 Infinite terms

Recently [CO90, DKP91, FW91, KKSdV91] the concept of rewriting has been extended to infinite terms. For example, $f(f(f(...)))$ is an infinite term t such that $f(t)$ and t are identical. We can associate terms with trees in an obvious way, and then an infinite term corresponds to an infinite tree. Another way to define an infinite term is as a sequence $t_0, t_1, t_2, ...$ of terms in which each t_{i+1} is an instance of t_i; then the "limit" of this sequence can be thought of as an infinite term. One problem with this definition is that an infinite term may correspond to more than one such sequence. To make the sequence unique, we can specify that t_0 is $\perp$ and each t_i is obtained from t_{i-1} by replacing each occurrence of $\perp$ by either a variable, a constant symbol, or a term of the form $f(\perp, \perp, ..., \perp)$. Note that if t is finite then after some point, all t_i will be identical and free of $\perp$. We can make the set of infinite terms into a metric space by defining the distance between two terms as $1/2^d$ where d is the depth of their shallowest difference. Formally, if s_i is the sequence defining s and t_i defines t then d is the minimum i such that s_i and t_i differ. With this definition, we can formally define what it means for an infinite sequence of finite or infinite terms to converge, and what is the limit.

Another approach is to define a partial ordering on terms, an *approximation ordering*, by $s \leq t$ iff t can be obtained from s by replacing finitely or infinitely many occurrences of $\perp$ by arbitrary terms. Then, infinite terms can be seen as least upper bounds of sequences t_i of terms such that for all i, $t_i \leq t_{i+1}$.

Substitutions extend to infinite terms simply by allowing replacements of variables by infinite terms. Then, rewriting extends to infinite terms by allowing such substitutions during matching. In this way, we can obtain rewrite sequences involving finite or infinite terms. Infinite rewrite sequences can have limits in this formalism so we may have confluence in that the limit does not depend on how the rewriting is done, so long as it is *fair*, that is, no redex persists indefinitely long without being chosen for rewriting. In this case, we can say that the rewrite sequence terminates even though it is infinite. We can say that a term rewriting system is ω-terminating if all fair infinite rewrite sequences have limits. In fact, we can have rewrite sequences longer than ω in this formalism, since it can be that the limit rewrites further to another term, and so on. Thus we can have rewrite sequences whose lengths are higher ordinals. This formalism permits us to model streams and infinite objects within the theory of term rewriting systems

The standard problems of rewriting appear in this framework: how to show that a system is ω-terminating, how to show it is confluent, and what can one say about the algebraic semantics of such systems. There is an interesting issue about termination: is the system $g(x) \rightarrow g(f(x))$ terminating? It has a limit $g(f(f(f(...))))$, so it is terminating according to our definition. However, the rewrites continue to occur near the top of the term; another possible definition is to say that a system is terminating only if the rewrites occur deeper and deeper in the term. For a discussion of these issues, see [CO90, DKP91, FW91, KKSdV91]. Note that outermost confluence may have an application here – if all outermost derivations have the same limit then lazy rewriting has a well defined semantics. So even if the system is not confluent, it may mean something for it to be outermost confluent; thus the sufficient conditions for outermost confluence given in section 12 may have an application to ω rewriting.

15.8 Conditional term rewriting

Sometimes equational systems are not expressive enough. For example, we may want to state that $s = t$ if some condition C is true. We call this a *conditional equation*. Many specifications are conditional, so conditional equations are natural for them. In these cases, we cannot use conventional term rewriting systems, since they correspond to sets of equations. Instead, we use *conditional term rewriting systems*, in which the rewrite rules have *conditions* attached, which must be true for the rewrite to occur. For example, a rewrite rule $s \rightarrow t$ with a condition C is expressed as

$$C \mid s \rightarrow t$$

where C is viewed as a *guard* for the rewrite rule $s \to t$. The conditions C may be of several different forms; for example, they may be logical formulas or equations or inequations. If C is a logical formula, the meaning is that s rewrites to t if C is true, but then there is the question of how to determine whether C is true, and even what it means.

There are various methods of defining the semantics of conditional term rewriting systems. For example, if we have

$$\begin{aligned} p(c) \quad &| \quad a \to b \\ not\ p(c) \quad &| \quad a \to b \end{aligned}$$

can we rewrite a to b? We might say yes, since either $p(c)$ is true or *not* $p(c)$ is true. We might say no, since neither condition can be proved. For a discussion of a logic-based semantics and several operational semantics for conditional systems, see [BDJ78]. Another semantics based on logic is given in [Pla87].

Note that the above rules can be expressed as

$$\begin{aligned} p(c) = true \quad &| \quad a \to b \\ not\ p(c) = true \quad &| \quad a \to b \end{aligned}$$

In this way, we can convert conditions C involving arbitrary formulas to conditions involving equations. However, the law of the excluded middle no longer holds; we do not have $x = true$ or $x = false$ for all x. This changes the semantics, of course. Using this idea, a Prolog clause $L : -L_1, L_2, ..., L_n$, meaning L_1 and L_2 and ... and L_n imply L, can be expressed as the conditional rewrite rule $L_1 = true, L_2 = true, ..., L_n = true \mid L = true$. This gives a connection between logic programming and conditional term rewriting. However, logic programming permits variables to be bound by unification, whereas conditional term rewriting typically uses matching instead, which is more restrictive. To simulate Prolog, something like "conditional narrowing" is needed. See [DP88] for one approach to conditional narrowing.

In this section, for simplicity, we will assume that C is a conjunction of equations. For example, we may have the rules

$$\begin{aligned} (x > y) = true \quad &| \quad max(x, y) = x \\ (x > y) = false \quad &| \quad max(x, y) = y \\ true \quad &| \quad (s(x) > s(y)) \to (x > y) \\ true \quad &| \quad (s(x) > 0) = true \\ true \quad &| \quad (0 > x) = false \end{aligned}$$

which are sufficient to define max and $>$ on terms of the form $s^i(0)$. If C is a conjunction of equations, then C can be evaluated by recursive conditional rewriting. This recursive rewriting invokes subconditions in a manner similar to the goal-subgoal structure of Prolog, making conditional term-rewriting systems an attractive combination of logic and functional programming paradigms.

We now discuss more precisely the syntax and semantics of conditional term rewriting systems.

In a rule of form $C \mid A$ we call C the *premiss* and A the *conclusion*. The semantics is that $C \supset A$. In general, C and A may be logical formulas, and may contain the equality predicate or the rewrite relation $\rightarrow$ and its variants. If C is a conjunction of (zero or more) equations and A is an equation, then we call this conditional rule a *positive conditional equation*. If C is a conjunction of equations and A is a rewrite rule (that is, of the form $s \rightarrow t$) we call the conditional rule a *semi-equational* rule. If C is a conjunction of statements of the form $s_i \downarrow t_i$ and A is a rewrite rule, we call the conditional rule a *join rule*. Corresponding to the three kinds of conditional rules, we have positive conditional systems, semi-equational systems, and join systems.

Interpreting the rule $C \mid A$ as $C \supset A$, interpreting $=$ as $\leftrightarrow^*$ for semi-equational systems, and using reasonable axioms for equality and the relations $\rightarrow$ and $\downarrow$, we obtain a concept of logical consequence for conditional term rewriting systems of the three types. (There are a number of other types, but these seem to be the most natural ones.) The most natural of these three from the declarative viewpoint are the positive conditional systems. The most natural from an operational point of view are the join systems. The semi-equational systems are in between.

We say a conditional system R is *noetherian* if there are no infinite sequences $t_1, t_2, t_3, \ldots$ of terms such that $t_1 \rightarrow_R t_2 \rightarrow_R t_3 \ldots$. This corresponds to the concept of termination for unconditional systems. However, the term "noetherian" is better suited to conditional systems, since each rewrite step involves the recursive evaluation of conditions. That is, to determine whether $s \rightarrow_R t$, it may be necessary to determine whether $u \rightarrow_R v$ for u and v obtained from the condition of some rule of R. It is possible for a system to be noetherian even when such recursive evaluation of conditions does not terminate; in such a case, the system might not be considered terminating in the usual sense of the term. In fact, it may happen that the rewrite relation $s \rightarrow_R t$ may be undecidable; that is, it may be undecidable for varying s and t whether $s \rightarrow_R t$.

The natural question, as for unconditional term rewriting systems, is what are the relations between these systems. In particular, we would like to decide the equational theory of positive conditional systems. One way to do this would be to find an equivalent join system with a noetherian and confluent rewrite relation, as in ordinary term rewriting systems. However, this is complicated by the fact that the rewrite relation itself can be undecidable for conditional systems.

For these reasons, it is convenient to define the concept of *decreasing systems*. Formally, a conditional system R is *decreasing* if there is a partial ordering $>$ such that 1) If $R \models s \rightarrow t$ then $s > t$ 2) $s > t$ if t is a proper subterm of s 3) $>$ is well founded, and 4) for all rules $C \mid l \rightarrow r$ in R, $l\sigma > s\sigma$ for all terms s in C appearing as arguments to $=$ or $\downarrow$. Note that the ordering $>$ does not have to have the replacement property; we may have $s > t$ and $f(s) < f(t)$. In addition, conditions 2) and 4) imply that all variables in C appear also in the left-hand side l. Also, if R is decreasing, then the rewriting relation is decidable, since conditions evaluated during the rewriting of l will involve terms smaller than l in the ordering. Also, it's possible for a conditional rewrite relation to be noetherian without being decreasing. In practice, simplification orderings $>$ may be used to show the decreasing property.

We now discuss criteria for confluence. Note that confluence in general is undecidable [BDJ78, Kap87], unlike the unconditional case. Critical pairs are defined similarly to the unconditional case, except that each critical pair may have an associated condition. Also, joinability of critical pairs is similar to the unconditional case, except that we are only interested in joinability when the associated condition is true. Thus the critical pairs are essentially triples (C, s, t) where C is a condition and s and t are terms, and we are interested in whether $s\Theta \downarrow t\Theta$ for Θ such that $C\Theta$ evaluates to *true*. For details, see for example [DOS87, DOS88]. Then one can show that semi-equational systems are confluent if they are noetherian and all critical pairs are joinable. Join systems are locally confluent if they are decreasing and all critical pairs are joinable. Thus they are confluent if they are in addition noetherian. Join systems are confluent if they are noetherian and all critical pairs are overlays (see section 12) and joinable. There are a number of other more specialized conditions for confluence.

In order to obtain decision procedures, we need to consider logical strengths of systems. The reason for this is as follow: Suppose we are given an equational system E (with equational conditions). We want to obtain an equivalent canonical system which can be used to decide the equality theory of E. One way to do this is to convert E into a join system R, and then complete R to obtain R' which is decreasing, noetherian, and confluent. Then we know that $R \models s = t$ iff $R' \models s = t$ by the way completion is done, and $R' \models s = t$ iff s and t have the same R'-normal form, since R' is noetherian and confluent. However, we do not know that $R' \models s = t$ iff $E \models s = t$, which is what we really want. The reason is that we changed the conditions in E to join assertions to obtain R. Therefore, the following theorem from [DOS88] is useful:

Theorem 15.8.1. *Let R be a confluent and noetherian join system and R' the correspoding semi-equational systems (with conditions changed to equations) and E the corresponding equational system (with right-hand sides changed to equations). Then the following are equivalent: (1) $E \models u = v$ (2) u and v have the same R-normal form (3) u and v have the same R'-normal form.*

If the conditions of the theorem hold, then R-reduction can be used to decide the theory of E. Note that typically E, when converteed to R, will not be confluent and noetherian. Therefore we have to complete R and do it in a way that does not change the corresponding equational theory E.

Completion for conditional systems is trickier than for unconditional systems. For some approaches, see [Gan91, Der91]. Recent work in conditional term rewriting may be found in proceedings of workshops in conditional term-rewriting [KJ87, KO90].

16 First-order theorem proving

The general theorem proving problem is, given a formula in some logical system, to decide if it is valid. Alternatively, we may want to decide if a formula is satisfiable, or if it is a logical consequence of a set of formulas. The discussion of rewriting is really concerned with theorem proving, although in a specialized context. For a discussion of general first-order theorem proving, see chapters ?? and ??. We now discuss applications of equational reasoning to first-order theorem proving.

16.1 Congruence closure

First we discuss a restricted case, that is, deciding whether $E \models s = t$ where E is a set (conjunction) of ground equations and s and t are ground terms. For example, we may want to decide whether $\{f^5(c) = c, f^3(c) = c\} \models f(c) = c$. This is a case in which rewriting techniques apply but another method is more efficient. The method is called *congruence closure* [NO80]; for some efficient implementations and data structures see [DST80]. The idea of congruence closure is essentially to use equality axioms, but restricted to terms that appear in E, including its subterms. For the above problem, we have the following derivation of $f(c) = c$, identifying equations $u = v$ and $v = u$:

1. $f^5(c) = c$ (given)
2. $f^3(c) = c$ (given)
3. $f^4(c) = f(c)$ (2, using equality replacement)
4. $f^5(c) = f^2(c)$ (3, using equality replacement)
5. $f^2(c) = c$ (1,4, transitivity)
6. $f^3(c) = f(c)$ (5, using equality replacement)
7. $f(c) = c$ (2,6, transitivity)

One can show that this approach is complete. It's immediate from Birkhoff's theorem that if $E \models s = t$ then there is a sequence of replacements of equals by equals relating s and t. For congruence closure, it would suffice to show that such a sequence exists in which all the intermediate terms appear in E. However, this is not always true, but congruence closure still works.

Another approach to this is to use the Knuth-Bendix method. In [GNP+88, Sny89] a polynomial time version of the Knuth-Bendix method is given for ground equational systems. For the above system, we would have the critical pair $(f^2(c), c)$ between the two equations in E, yielding the equation (rewrite rule) $f^2(c) = c$. Superposing this with $f^3(c) = c$, we get the critical pair $(c, f(c))$ and the equation $f(c) = c$.

16.2 Paramodulation

In a more general setting, the need for demodulation (i.e., rewriting) in theorem proving arises from the need to avoid generating many terms like $x * 1$, $x * 1 * 1$, $1 * x * 1$, et cetera. Without some method of simplifying such terms, a theorem prover becomes bogged down with many equivalent terms. There is another operation, called *paramodulation*, which is more general than demodulation and is needed for completeness in theorem proving without the equality axioms. Paramodulation is analogous to finding a critical pair, except that it involves two formulas which may be more general than equations. One of these must have a sub-formula involving equality.

For this section, we need some definitions for first-order logic. An *atom* is a predicate symbol followed by a list of terms, as, $P(x, f(a))$. A *literal* is an atom or an atom preceded by a negation. A *clause* is a set of literals, signifying their disjunction. It is well known that there is a satisfiability-preserving translation of any first-order formula into a set (conjunction) of clauses; many theorem provers operate on such clause-form formulas for simplicity and efficiency reasons. Paramodulation is defined on clauses, as follows:

From clauses $\{L_1(s), L_2, L_3, ..., L_m\}$ and $\{u = v, M_2, M_3, ..., M_n\}$ to derive a paramodulant $\{L_1(v\sigma), L_2\sigma, ..., L_m\sigma, M_2\sigma, ..., M_n\sigma\}$, where s is a subterm of L_1 and s and u are unifiable with most general unifier σ. For example, from $\{P(f(x)), Q(g(x))\}$ and $\{f(a) = c, R(b)\}$ we can infer $\{P(c), Q(g(a)), R(b)\}$, obtained by replacing x by a in the first clause and replacing $f(a)$ by c using the second clause. For this we assume that the two given clauses have no common variables; if so, they are renamed beforehand to be disjoint. Note that this generalizes rewriting, in that a substitution is applied to s as well as to the equation $u = v$. Also, this generalizes rewriting because the second given clause has other literals in addition to the equation $u = v$. In addition, the variable restriction for left and right hand sides of rewrite rules is relaxed. One can show that a paramodulant of two clauses is a logical consequence of those clauses. There may be more than one paramodulant of two clauses.

The first proof that paramodulation is complete is due to Brand [Bra75]. He shows that if a set of clauses is unsatisfiable, there is a proof of the empty clause using resolution and paramodulation, if the equality axiom $\{x = x\}$ is included in the input. The other equality axioms are not needed. His proof works by considering a "pulled-out" form of the input in which new equality literals are added to clauses, and resolving on these literals corresponds to paramodulation on the original clauses. For example, the transformed form of the clause $\{P(g(f(x)))\}$ would be $\{f(x) \neq y, g(y) \neq z, P(z)\}$. He shows that the transformed clauses, together with $\{x = x\}$, are unsatisfiable iff the original clauses, together with the equality axioms, are unsatisfiable. Thus, resolution is sufficient to derive the empty clause from the transformed clauses, if they are unsatifiable. However, since some resolutions on the transformed clauses correspond to paramodulation on the original clauses, this shows that resolution and paramodulation suffice to derive the empty clause from the original clauses. His proof is interesting, apart from the completeness of paramodulation, because of the restriction on paramodulation that it implies. The paramodulations obtained from Brand's transformation are only done on term structure near the top of terms, that is, term structure that was originally present in the clauses. The idea of restricting critical pairs to term structure that was originally present in the input clauses is very attractive, especially on problems where the input equations are small but the equations produced during inference are large. Few people appreciated that Brand's proof implied that this restriction on paramodulation is complete. We have implemented such a restriction in our extension of the clause linking theorem prover CLIN [LP90] to equality. In [BGLS91], Snyder et al have independently shown the completeness of this method of paramodulation, which they call *basic paramodulation*, in analogy to the basic narrowing strategy which will be presented later.

Their proof allows more restrictions on paramodulation such as those explained below involving a termination ordering. They show completeness of paramodulation and resolution in combination with rewriting, which Brand did not show.

The completeness of this restriction also gives insight into the Knuth-Bendix method. In particular, Brand's transformation on pure equational systems corresponds to a version of the Knuth-Bendix method where rewriting (simplification) of previously derived rewrite rules is not done, and where critical pairs can be obtained by superposing the left-hand side of one rule with the right-hand side of another rule. Also, this version of the Knuth-Bendix method only permits superposition to be done near the top of the rules, on term structure that was originally present in the input clauses, not introduced during inference. Therefore, it seems attractive to see what versions of the Knuth-Bendix method can be devised which retain this restriction, but allow more rewriting (simplification) of previously derived rules.

Since Brand's proof, there have been other, possibly simpler proofs of the completeness of resolution and paramodulation. One was by Peterson [Pet83], using semantic trees. This proof showed that it is complete to use a version of paramodulation in which equations are only used in one direction, specified by a termination ordering on terms; if $u = v$ is an equation and $u > v$ in the termination ordering, then the equation only needs to be used in the direction $u \rightarrow v$, replacing instances of u by instances of v. Brand's proof does not mention termination orderings in general Later, Hsiang and Rusinowitch [HR91] found a more general proof, using transfinite semantic trees, allowing more general termination orderings. Others have derived related results (Zhang and Kapur [ZK88] and Bachmair and Ganzinger [BG90]); the proof of Bachmair and Ganzinger does not use transfinite semantic trees. These proofs generally show that paramodulation with some significant refinements is still complete. These refinements involve choosing a literal to paramodulate into, choosing a direction in which to use the equality literal, and so on. Generally, there is an ordering on literals, and it is only necessary to choose literals that are maximal in this ordering. Also, there is an ordering on terms, and an equation $s = t$ need only be used in the direction of replacing the side larger in the ordering by the side smaller in the ordering. This can be extended, as for unfailing completion, to look at the instance of the equation used, which may be orderable even if the equation itself is not.

These results typically show that paramodulation and resolution together are complete. However, in a sense this may be beside the point. It may turn out that other theorem proving methods are more efficient than resolution; we give partial evidence in that direction in [LP90] for the clause linking method. Similar remarks may apply to the connection calculi discussed in chapter **??**. If this is true, then it may be more important to show that some version of paramodulation is complete in combination with these other strategies, such as the connection calculi. Still, there will probably be some problems for which resolution and paramodulation are the best combination of strategies.

A fairly straightforward approach to paramodulation is to simulate it by transforming the input in a manner similar to that of Brand [Bra75]. This eliminates the need for a separate paramodulation operation. This simulation tends to reduce redundancy because the paramodulations at different positions are considered separately, reducing the number of orders in which paramodulation can be applied to disjoint subterms. Also, this transformation is more compatible with the clause linking method [LP90] than conventional paramodulation. Further, as mentioned above, this means that paramodulations are restricted to be near the top of each term, rather than occurring anywhere in the term. However, using the transformed set of clauses is slower for each inference than paramodulation, since there is the overhead of simulating paramodulation by resolution or some equivalent inference rule. Another equality transformation is given by Digricoli and Harrison [DH86], who transform each assertion (clause) as it is generated, but generally produces shorter clauses than those in Brand's method.

There is some reason to believe that it is never necessary to perform paramodulation inside Skolem functions [McC89b]. This is an interesting refinement that has the potential for significant reductions in the search space size.

Some theorem provers convert an arbitrary set of clauses into a conditional term rewriting system, and then use methods specialized to such systems. For example, see [ZK88]. But, to convert a clause to a conditional rule, one has to choose how to do it, since there is a choice which literals get put in the conditions. This involves user guidance, which has advantages and disadvantages. We would like to be able to get along without such guidance if possible, but it does provide a means for the user to guide the search if he or she is an expert in theorem proving.

Another application of rewriting to theorem proving is in the simplification rules at clause level. For example, a clause $\{P(a), Q(a), R(a)\}$ can be simplified to $\{Q(a), R(a)\}$ if there is a clause $\{\neg P(x)\}$. This can be viewed as rewriting in which any instance of the literal $P(x)$ is replaced by $false$ and the resulting clause simplified. This simple idea is sometimes very helpful. Still another application is rewriting at the formula level, to replace formulas by their definitions. This is quite helpful in simple set theory problems; for one application of this idea to Von Neumann-Bernays-Godel set theory see [PP91]. In fact, all of first order logic can be embedded in equational logic; one approach to this is the representation of Hsiang [Hsi85] and another is that of McKenzie [McK75]. Since all of Von Neumann-Bernays-Godel set theory can be encoded in first-order logic, all of Von Neumann-Bernays-Godel set theory can be encoded in pure equational logic. However, it's not clear that this leads to efficient theorem provers, because the symmetries of the connectives and quantifiers and set operators are obscured in this translation.

An interesting research problem is to extend paramodulation and its refinements to strategies using E-unification for various sets E of equations. Another research problem is to extend equational methods to back chaining and set of support strategies. The Knuth-Bendix method is basically a forward chaining method that is not sensitive to the theorem being proved. For one approach to back chaining and equality, see [DS88]. Yet another problem is to avoid the generation of huge terms. Our clause linking prover at times generated terms over 150,000 characters long by rewriting with a terminating system of rules. Many well-known provers deal with this problem by setting a bound on the number of rewrites permitted on a single clause, or by setting an arbitrary size bound. However, these methods are not altogether satisfactory, and something more systematic is needed. What we did in our clause linking implementation is to first rewrite each clause using rewrite rules that cannot possibly increase the length (size) of a clause. Later, arbitrary rewriting is done. In our prover, this works well because the prover automatically sets a size bound that decreases with the work done.

16.3 E-unification algorithms

When the set of axioms in a theorem to be proved includes a set E of equations, it is often better to use specialized methods than general theorem proving techniques. For example, if the binary infix operator $*$ is associative and commutative, we may generate many equivalent terms $x * (y * z)$, $y * (x * z)$, $y * (z * x)$, etc. These cannot be eliminated by rewriting since none is simpler than the others. Even the idea of using unorderable equations as rewrite rules when the applied instance is orderable, won't help. One approach to this problem is to incorporate a general E-unification algorithm into the theorem prover. Plotkin [Plo72] first discussed this general concept and showed its completeness in the context of theorem proving. This permits us to keep only one representative of each E-equivalence class, significantly reducing the number of formulas retained. E-unification is also known as semantic unification, which may be a misnomer since no semantics (interpretation) is really involved. For a discussion of E-unification, see chapter **??**. The general idea is that if E is a set of equations, an E-unifier of two terms s and t is a substitution Θ such that $E \models s\Theta = t\Theta$, and a most general E-unifier is an E-unifier that is as general as possible in a certain technical sense relative to the theory E. For some theories, there may be at most one most general E-unifier, and for others, there may be more than one, or even infinitely many, most general E-unifiers.

An important special case, already mentioned above in the context of term-rewriting, is associative-commutative (AC) unification. In this case, if two terms are E-unifiable, then there are at most finitely many most general E-unifiers, and there are algorithms to find them that are usually efficient in practice. The well-known algorithm of [Sti81] essentially involves solving Diophantine equations and finding a basis for the set of solutions and finding combinations of basis vectors in which all variables are present. This can sometimes be very time consuming; Domenjoud [Dom90] shows that the two terms $x + x + x + x$ and $y_1 + y_2 + y_3 + y_4$ have more than 34 billion different AC unifiers. In this case, one might say that the problem is really hard, and that the difficulty of the AC unification simply reveals that. Another approach is to reduce the work involved in AC unification. For example, we might consider resource bounded AC unification, that is, finding all unifiers within some size bound. This might reduce the number of unifiers in cases where many of them are very large. Another idea is to consider "optional variables," that is, variables that may or may not be present. If x is not present in the product $x * y$ then this product is equivalent to y. This greatly reduces the number of AC unifiers, and has been studied by [Dom91] but possibly not implemented yet. This permits one to represent a large number of solutions compactly, but requires one to keep track of optionality conditions.

16.4 Rule-based unification

Unification can be viewed as equation solving, and therefore is part of theorem proving or possibly logic programming. This approach to unification permits conceptual simplicity and also is convenient for theoretical investigations. For example, unifying two literals $P(s_1, s_2, ..., s_n)$ and $P(t_1, t_2, ..., t_n)$ can be viewed as solving the set of equations $\{s_1 = t_1, s_2 = t_2, ..., s_n = t_n\}$. Given such a set of equations, there are a collection of rules for transforming it to either obtain a most general unifier or detect non-unifiability. For example, one rule replaces an equation $f(u_1, u_2, ..., u_n) = f(v_1, v_2, ..., v_n)$ by the set of equations $\{u_1 = v_1, u_2 = v_2, ..., u_n = v_n\}$. Another rule detects non-unifiability if there is an equation of the form $f(...) = g(...)$ for distinct f and g. Another rule detects non-unifiability if there is an equation of the form $x = t$ where t is a term properly containing x. With a few more such rules, we can obtain a simple unification algorithm that will terminate with a set of equations representing a most general unifier. For example, the set of equations $\{x = f(a), y = g(f(a))\}$ would represent the substitution $\{x \leftarrow f(a), y \leftarrow g(f(a))\}$. This approach has also been extended to E-unification for various equational theories E. For a survey of this aproach, see [JK91].

17 Logic programming and narrowing

Prolog-style Horn clause logic programming [Llo87] is based on first-order theorem proving. The idea is to provide constructive proofs of theorems of the form $(\exists\overline{x})P(\overline{x})$. Since the proof is constructive, an $\overline{x}$ satisfying P can be mechanically constructed from the proof. No concept of term rewriting or equality is intrinsic to Horn clause logic programming. (Of course, equational logic programming [O'D85] is based entirely on sets of equations.) It is desirable, therefore, to incorporate some form of equality and term rewriting in Horn clause logic programming. More generally, there is a desire to combine the attractive features of logic programming and functional programming in a useful and semantically clean way. Many such combinations have been proposed; see for example DeGroot and Lindstrom [DL86]. We tend to emphasize those that are closest to term rewriting systems. Some of these combinations make use of the idea of *narrowing*. Narrowing was proposed in [Fay79], refined by Hullot [Hul80], and applied to logic and functional programming by Reddy [Red85]. Narrowing for conditional term rewriting systems and its application to logic and functional programming was studied by Goguen and Meseguer [GM84] and Dershowitz and Plaisted [DP88]. A language combining narrowing with a kind of clausal superposition (analogous to Knuth-Bendix completion) is presented in [Fri85]. Conditional narrowing is attractive because it combines rewriting with logic programming, and also permits a recursive condition evaluation which is much like the subgoal evaluation of Prolog. Non-logical features like cut do not seem as essential in this approach, but some efficiency is lost as compared to Prolog. Jayaraman and Silbermann developed the EqL language [JS86] which implements *object refinement*, having some similarities to narrowing, but with more user control. Another language with rewriting and nondeterminism is the SEL language of Jayaraman and Plaisted [JP87]. This is similar to Horn clause logic programming in the nondeterminism achieved by non-confluent term rewriting systems but differs in the use of sets to model nondeterminism.

An early implementation of narrowing with some refinements is given in Josephson and Dershowitz [JD86]. The concept of narrowing is partially implicit in paramodulation in theorem proving, however. More recently, Nutt, Rety, and Smolka [NRS89] and others have discussed the completeness of various combinations of refinements to narrowing. A good, short discussion of narrowing in the context of rule-based unification is given in [JK91].

The idea of narrowing is, given a term rewriting system R, to find a most general instance $s\Theta$ of a term s that $s\Theta \rightarrow_R^* true$. For example, s may represent a query, and the variables are instantiated by Θ to values that make the query true. In this case, R rewriting corresponds to evaluating the query, whether it is true or not. In this way, the logic programming idea of finding values for variables is extended to the functional programming framework, providing a partial combination of logic and functional programming. Each narrowing step is a rewriting step that uses unification instead of matching, to generate a most general substutition permitting the rewrite rule to apply. For example, we can narrow the term $g(f(a,x))$ using the rewrite rule $f(a,h(a)) \rightarrow c$ to yield the term $g(c)$. The substitution $\{x \mapsto h(a)\}$ has been applied to the term $g(f(a,x))$ to enable the rewrite rule to apply. In this way, we can find most general instances of a term reducing to true (or having some specified value). Formally, we have the following definition.

Definition 17.0.1. Given a term s and a term rewriting system R, $s \rightsquigarrow_{R,\Theta} t$ (s narrows to t) if s has a non-variable subterm u and there is a rewrite rule $l \rightarrow r$ in R and u and l are unifiable with most general unifier Θ and t is obtained from $s\Theta$ by replacing $u\Theta$ by $r\Theta$.

This definition differs slightly from others in the literature. It is customary to omit Θ from the notation and to specify which rewrite rule is used, rather than R as a whole. However, the idea is essentially the same. We now want to state precisely in what sense narrowing captures the most general instance of a term reducing to another term. We have the following result, similar to a result in [Hul80], where $\rightsquigarrow_R^*$ indicates zero or more steps of $\rightsquigarrow_{R,\Theta}$ for various Θ. Thus, if $r \rightsquigarrow_{R,\Theta} s$ and $s \rightsquigarrow_{R,\Gamma} t$ then we can say that $s \rightsquigarrow_R^* t$.

Theorem 17.0.2. *Suppose R is a canonical term rewriting system and $s\rho \rightarrow^* t$. Then there exist terms u and v such that $s \rightsquigarrow_R^* u$ and $t \rightarrow_R^* v$ and v is an instance of u.*

This theorem is hard to understand without some examples. Suppose s is $f(x,x)$ and ρ replaces x by a. Suppose R is the rule $a \rightarrow b$. Suppose t is $f(a,b)$. Then $s\rho \rightarrow t$ because $f(a,a) \rightarrow f(a,b)$. However, s does not narrow at all since x is a variable and cannot be narrowed. Thus u is s and v is $f(b,b)$ which is an instance of u. The problem is that the terms in $s\rho$ that replace variables in s, may rewrite to different terms in t. Since narrowing does not replace variables, it may not be possible to mimic these rewrites by narrowing s. However, it is possible to rewrite t to obtain a term in which all these variables have been replaced by the same term, since R is canonical, hence confluent. In this way, we obtain an instance of a term to which s can narrow.

From a narrowing derivation $s \rightsquigarrow_R^* u$ we want to obtain a substitution θ such that $s\theta \rightarrow_R^* u$. This substitution gives the values of the variables in s permitting a rewriting to u, which is as general as v or more general. We obtain θ by composing all the substitutions in the narrowing derivation, and restricting the result to the variables that appear in s. We write $s \rightsquigarrow^*_{R,\theta} u$ to indicate that there is a narrowing derivation from s to u in which θ is the composition of the substitutions used, restricted to s.

The proof of the above result goes like this: Let s' be $s\rho$ with subterms corresponding to variables in s, reduced to normal form. This is possible because R is canonical. Note that s' is an instance of s since repeated occurrences of variables in s correspond to repeated subterms of t which reduce to the same normal form in s'. Also, $s\rho \rightarrow^*_R s'$ and $s\rho \rightarrow^*_R t$. Since R is canonical, there is a term v such that $s' \rightarrow^*_R v$ and $t \rightarrow^*_R v$. Now, consider the rewriting of s' to v. Each rewriting step will involve some term corresponding to a non-variable subterm of s, since the subterms of s' corresponding to variables in s, are in normal form. Thus, each rewriting on s' can be simulated by a narrowing step on s. In this way, we get a sequence of narrowing steps on s that lead to a term u having v as an instance. This completes the sketch of a proof.

Note that this narrowing sequence has a particular form. In particular, certain subterms of s and terms obtained from s by narrowing, are never narrowed themselves. These are the variables of s and the terms obtained by instantiating these variables in later narrowing steps. These never need to be narrowed, since in the rewriting from s', these correspond to terms in normal form that cannot be rewritten (each narrowing step from s corresponds to a rewriting step from s'). In this way we obtain what is called *basic narrowing*. Another restriction to narrowing is *normal narrowing*, in which each term is rewritten to normal form before narrowing. Their combination is also used, and satisfies the same property given above, provided that basic narrowing is redefined so that allowable subterms for narrowing are modified to account for the normalization steps; see [R$\acute{8}$7]. In general, by using a particular rewriting strategy on s', such as innermost rewriting, we get completeness of a restricted version of narrowing.

The following result is an immediate corollary:

Corollary 17.1. Suppose R is a canonical term rewriting system and $s\rho \rightarrow^* t$ and t is in normal form. Then there exist term u such that $s \rightsquigarrow^*_R u$ and t is an instance of u.

In fact, it's not necessary to assume that t is in normal form. It's only necessary that the subterms of $s\rho$ corresponding to variables of s, are in normal form, that is, that ρ is normalized.

We note that confluence is needed for the results. For a counter-example, let R consist of the rules $a \rightarrow b$ and $a \rightarrow c$. Suppose s is $f(x, x)$ and $s\rho$ is $f(a, a)$. Suppose t is $f(b, c)$. Then $s\rho \rightarrow^*_R t$ and t is in normal form but s cannot be narrowed and t is not an instance of s.

If terms are linear then the result can be generalized:

Theorem 17.0.3. *Suppose R is a term rewriting system in which the left and right hand sides of all rules are linear (no repeated variables). Suppose s is linear and $s\rho \rightarrow^* t$. Then there exist a term u such that $s \rightsquigarrow^*_R u$ and t is an instance of u.*

Note that we do not require R to be confluent or terminating. The proof makes use of the fact that all terms in the rewriting of $s\rho$ to t are linear, and therefore each rewriting step can be simulated by a narrowing step from s, or possibly by a change in the substitution relating the two derivations.

Narrowing can be used for unification. To see this, suppose s and t are E-unifiable, that is, there are instances s' and t' of s and t which are E-equal. Suppose R is a canonical term rewriting system equivalent to E. Then s' and t' both rewrite to the same normal form u using R. Even more, the equation $s' = t'$ rewrites to $u = u$. Thus the equation $s = t$ narrows to some equation $v = w$ having $u = u$ as an instance. Let α be the substitution of this narrowing. Let β be the most general unifier of v and w (which must exist since v and w both have u as an instance). Then $\alpha\beta$ (the composition of α and β) is an E-unifier of s and t. One can show that this process yields all most general such substitutions, and thus we obtain a unification algorithm for any equational theory having an equivalent canonical term rewriting system. There are canonical systems for which most general E-unifiers do not exist; see [FH86]. Of course, this is of limited value since narrowing may not terminate, that is, there may be infinitely many narrowings of a given term, so it may not be possible to enumerate them all in finite time. Some kind of combination of narrowing and unfailing completion can be used for unification in equational theories having no equivalent finite canonical term rewriting system.

We would like to have rewriting (demodulation) also in logic programming, as well as narrowing. One problem with both techniques is inefficiency, since we have to look at the entire term structure to see if any rewriting or narrowing can be done. Horn-clause logic programming just unifies at the top level of each literal, which requires less search.

Another way to incorporate equality into logic programming is to use some equality transformation like that of Brand [Bra75] together with the usual Prolog execution mechanism. This transformation eliminates the need for the equality axioms and leads to a fairly efficient treatment of equality. Such transformed clauses are called *flat* Horn clauses in the logic programming community and the proofs are called *surface* deductions. This idea was introduced in the logic programming community by Barbuti, Bellia et al [BBLM86] and by Tamaki [Tam84]; see also [CP85]. Using this idea, Bosco and Giovannetti [BG86] have developed a clean integration of logic and functional programming. They also obtain some higher order expressiveness by using the *apply* function, so that *apply*(f, x) represents $f(x)$. This permits f to be a variable, so that first-order logic can be used to simulate some higher order features.

18 Acknowledgements

I would like to thank Nachum Dershowitz, Steffen Hoelldobler, and Deepak Kapur for their comments on preliminary versions of this chapter, as well as Geoffrey Alexander, Heng Chu, James Riely, Bill Yakowenko, and the other students of my rewriting seminar who pointed out errors in the chapter.

References

[Bar84] H. Barendregt. *The Lambda-Calculus, its syntax and semantics.* North-Holland, Amsterdam, 1984. Studies in Logic and the Foundation of Mathematics.

[BBLM86] 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, 1986.

[BD86] Leo Bachmair and N. Dershowitz. Commutation, transformation, and termination. In J.H.Siekmann, editor, *Proceedings of the Eighth International Conference on Automated Deduction*, pages 5–20, 1986.

[BDH86] Leo Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *Proceedings of the Symposium on Logic in Computer Science*, pages 346–357, 1986.

[BDJ78] D. Brand, J.A. Darringer, and W.J.Jr. Joyner. Completeness of conditional reductions. Technical Report RC 7404, IBM Research Center, Yorktown Heigghts, N.Y., December 1978.

[BDP89] Leo Bachmair, N. Dershowitz, and D. Plaisted. Completion without failure. In Hassan Ait-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures 2: Rewriting Techniques*, pages 1–30, New York, 1989. Academic Press.

[BG86] P.G. Bosco and E. Giovannetti. Ideal: an ideal deductive applicative language. In *Proceedings of the Third IEEE symposium on Logic Programming*, pages 89–94, September 1986.

[BG90] Leo Bachmair and Harold Ganzinger. On restrictions of ordered paramodulation with simplification. In Mark Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, pages 427–441, New York, 1990. Springer-Verlag.

[BGLS91] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and basic strict superposition. submitted, 1991.

[Bir35] G. Birkhoff. On the structure of abstract algebras. *Proc. Cambridge Philos. Soc.*, 31:433–454, 1935.

[BKN85] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. In *First International Conference on Rewriting Techniques and Applications*, pages 417 – 429, May 1985.

[Boo87] R.V. Book. Thue systems as rewriting systems. *Journal of Symbolic Computation*, 3((1,2)):39–68, February/April 1987.

[BP85] L. Bachmair and D. Plaisted. Termination orderings for associative-commutative rewriting systems. *J. Symbolic Computation*, 1:329–349, 1985.

[BPW89] T. Baird, G. Peterson, and R. Wilkerson. Complete sets of reductions modulo associativity, commutativity, and identity. In *Proceedings of the 3rd International Conference* *on rewriting techniques and applications*, pages 29–44, 1989. Lecture Notes in Computer Science, Vol. 355.

[Bra75] D. Brand. Proving theorems with the modification method. *SIAM J. Comput.*, 4:412–430, 1975.

[Buc87] B. Buchberger. History and basic features of the critical-pair/completion procedure. *J. of Symbolic Computation*, 3(1,2):3–38, February/April 1987.

[Bur88] H.-J. Burckert. Solving disequations in equational theories. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 517–526, 1988.

[CF58] H.B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[Chc90] E.A. Chchon. Abstract ordering structures and computational complexity. Technical Report CSD-TR-621, University of London, May 1990.

[Che81] P. Chew. Unique normal forms in term rewriting systems with repeated variables. In *Proceedings of the 13th Annual Symposium on the Theory of Computing*, pages 7–18, 1981.

[CL73] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.

[CL89] H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7(3 & 4):371–426, 1989. Special issue on unification. Part one.

[CO90] Y. Chen and M.J. O'Donnell. Infinite terms and infinite rewritings. In *2nd International Workshop on Conditional and Typed Rewriting Systems*, pages 45–52, 1990. to appear in Lecture Notes in Computer Science.

[CP85] P. T. Cox and T. Pietrzykowski. Surface deduction: a uniform mechanism for logic programming. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 220–227, 1985.

[Dau89] M. Dauchet. Simulation of turing machines by a left-linear rewrite rule. In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, pages 109–120, 1989. Lecture Notes in Computer Science 355.

[Der80] N Dershowitz. On representing ordinals up to γ_0. Unpublished note, 1980.

[Der82] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[Der84] N Dershowitz, 1984. personal communication.

[Der85] N. Dershowitz. Computing with rewrite systems. *Information and Control*, 64((2/3)):122–157, (May/June) 1985.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[Der91] N. Dershowitz. Canonical sets of horn clauses. In *Proceedings of ICALP '91*, pages 267–278, July 1991. Lecture Notes in Computer Science volume 510.

[DH86] V. Digricoli and M. Harrison. Equality-based binary resolution. *J. Assoc. Comput. Mach.*, 33(2):253–289, April 1986.

[DHJP83] Nachum Dershowitz, J. Hsiang, N. Josephson, and David A. Plaisted. Associative-commutative rewriting. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 940–944, August 1983.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, Amsterdam, 1990.

[DJ91] Nachum Dershowitz and Jean-Pierre Jouannaud. Notations for rewriting. *Bulletin of the European Association of Theoretical Computer Science*, (43):162–172, February 1991.

[DKP91] N. Dershowitz, S. Kaplan, and D. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite. *Theoretical Computer Science*, 83:71–96, 1991.

[DL86] D. DeGroot and G. Lindstrom. *Logic programming: functions, relations, and equations.* Prentice-Hall, Englewood Cliffs, N.J., 1986.

[Dom90] E. Domenjoud. Number of minimal unifiers of the equation $\alpha x_1 + ... + \alpha x_p =_{AC} \beta y_1 + ... + \beta y_q$. *Journal of Automated Reasoning*, 1990. Accepted for publication.

[Dom91] E. Domenjoud. Ac-unification through order-sorted ac1-unification. Technical report, 1991. Lecture Notes in Computer Science volume 488.

[DOS87] N. Dershowitz, M. Okada, and G. Sivakumar. Confluence of conditional rewrite systems. In *Proceedings of the First International Workshop on Conditional Rewriting*, July 1987.

[DOS88] N. Dershowitz, M. Okada, and G. Sivakumar. Canonical conditional rewrite systems. In *Proceedings of the Ninth Conference on Automated Deduction*, pages 538–549, May 1988. Lecture Notes in Computer Science volume 310.

[DP88] Nachum Dershowitz and D. Plaisted. Equational programming. In J.E.Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11*, pages 21–56. Oxford University Press, 1988.

[DS88] Nachum Dershowitz and G. Sivakumar. Goal-directed equation solving. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 166–170, August 1988.

[DST80] P.J. Downey, R. Sethi, and R. Tarjan. Variations on the common subexpression problem. *Journal of the Assoc. of Comput. Mach.*, 27(4):758–771, 1980.

[DTHL87] M Dauchet, S. Tison, T Heuillard, and P. Lescanne. Decidability of the confluence of ground term rewriting systems. In *Proceedings of the Second Symposium on Logic in Computer Science*, pages 353–359, 1987.

[Fay79] M. Fay. First order unification in equational theories. In *Proceedings of the 4th Workshop on Automated Deduction*, pages 161–167. Springer-Verlag, 1979. Vol. 87 of *Lecture Notes in Computer Science*, Springer, Berlin.

[FH86] F. Fages and G Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, 43:189–200, 1986.

[Fit90] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, New York, 1990.

[Fri85] L. Fribourg. Slog: A logic programming language interpreter based on clausal superposition and rewriting. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 172–184, 1985.

[FW91] W.M. Farmer and R.J. Watro. Redex capturing in term graph rewriting. In *Proceedings of the 4th International Conference on rewriting techniques and applications*. Springer-Verlag, 1991. Lecture Notes in Computer Science volume 488.

[Gal86] Jean H. Gallier. *Logic for Computer Science: Foundations of Automated Theorem Proving*. Harper and Row, New York, 1986.

[Gan91] H. Ganzinger. A completion procedure for conditional equations. *Journal of Symbolic Computation*, 11:51–81, January 1991.

[GKM83] J.V. Guttag, D. Kapur, and D. Musser. On proving uniform termination and restricted termination of rewriting systems. *SIAM J. Comput.*, 12:189–214, 1983.

[GM81] J.A. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *SIGPLAN Notices*, 16(7):24–32, July 1981.

[GM84] J.A. Goguen and J. Meseguer. Equality, types, modules and (why not?) generics for logic programming. *J. Logic Programming*, 1(2):179–210, 1984.

[GNP+88] Jean Gallier, P. Narendran, D. Plaisted, S. Raatz, and W. Snyder. Finding canonical rewriting systems equivalent to a finite set of ground equations in polynomial time. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 182 – 196, May 1988.

[Gog80] J.A. Goguen. How to prove algebraic inductive hypotheses without induction. In *Proceedings of the Fifth International Conference on Automated Deduction*, pages 356–373, July 1980.

[Gra47] A.A Grau. Ternary boolean algebra. *Bulletin of American Math Soc.*, 53(6):567–562, June 1947.

[GV88] D.Y. Grigor'ev and N.N Vorobjov. Solving systems of polynomial inequalities in subexponential time. *J. Symbolic Computation*, 5:37–64, 1988.

[HH80] G. Huet and J.M. Hullot. Proofs by induction in equational theories without constructors. In *21st Annual Symposium on Foundations of Computer Science*, pages 96 – 107, 1980.

[HH87] S. Heilbrunner and S. Hölldobler. The undecidability of the unification and matching problem for canonical theories. *Acta Informatica*, 24:157–171, 1987.

[HL78] G. Huet and D. Lankford. On the uniform halting problem for term rewriting systems. Technical Report Rapport Laboria 283, IRIA, Le Chesnay, France, 1978.

[HL79] G. Huet and J.-J. Levy. Call by need computations in non-ambiguous linear term rewriting systems. Technical Report Rapport 359, INRIA, 1979.

[HL89] D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations (preliminary version). In *Proceedings of the 3rd International Conference on rewriting techniques and applications*, pages 167–177, April 1989. Lecture Notes in Computer Science vol. 355.

[HN52] G. Higman and B.H. Neumann. Groups as groupoids with one law. *Publ. Math. Debrecen*, 2:215–221, 1952.

[HO79] C.M. Hoffmann and M.J. O'Donnell. Interpreter generation using tree pattern matching. In *Proceedings of 6th Annual Symposium on Principles of Programming Languages*, pages 169–179, 1979.

[Hof90] D. Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. In H. Kirchner and W. Wechler, editors, *Proceedings of the Second Conference on Algebraic and Logic Programming*, pages 347–358, Nancy, France, October 1990. Vol. 463 of *Lecture Notes in Computer Science*, Springer, Berlin.

[HR91] J. Hsiang and M Rusinowitch. Proving refutational completeness of theorem-proving strategies: the transfinite semantic tree method. *J. Assoc. Comput. Mach.*, 38(3):559–587, July 1991.

[Hsi85] J. Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, 25:255–300, March 1985.

[Hue80] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. Assoc. Comput. Mach.*, 27(4):797–821, 1980.

[Hue81] G. Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *J. Comput. Systems Sci.*, 23(1):11–21, 1981.

[Hul80] J.M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proceedings of 5th Conference on Automated Deduction*, volume 87, pages 318–334. Springer Verlag, Lecture Notes in Computer Science, July 1980.

[JD86] N. A. Josephson and N. Dershowitz. An efficient implementation of narrowing: the rite way. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 187–197, 1986.

[Jea80] J. Jeanrond. Deciding unique termination of permutative rewrite systems: Choose your term algebra carefully. In W. Bibel and R. Kowalski, editors, *Proceedings of 5th*

Conference on Automated Deduction, volume 87. Springer Verlag, Lecture Notes in Computer Science, 1980.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Comput.*, 15:1155–1194, November 1986.

[JK91] Jean-Pierre Jouannaud and Claude 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. To appear.

[JL82] J.-P. Jouannaud and P Lescanne. On multiset orderings. *Information Processing Letters*, 15:57–63, 1982.

[JLR82] J.-P. Jouannaud, P. Lescanne, and F. Reinig. Recursive decomposition ordering. In *Proceedings of the Second IFIP Workshop on FOrmal Description of Programming Concepts*, pages 331–348, 1982.

[JO91] Jean-Pierre Jouannaud and Mitsuhiro Okada. Satisfiability of systems of ordinal notations with the subterm property is decidable. In *Proceedings of the Eighteenth EATCS Colloquium on Automata, Languages and Programming*, Madrid, Spain, July 1991. Vol. 510 in *Lecture Notes in Computer Science*, Springer, Berlin.

[JP87] B. Jayaraman and D. Plaisted. Functional programming with sets. In *Third International Conference on Functional Programming and Computer Architecture*, pages 194–210, September 1987.

[JS86] B. Jayaraman and F.S.K. Silbermann. Equations, sets, and reduction semantics for functional and logic programming. In *1986 ACM Symposium on LISP and Functional Programming*, pages 320–331, 1986.

[Kap87] S. Kaplan. Simplifying conditional term rewriting systems: Unification, termination, and confluence. *Journal of Symbolic Computation*, 4(3):295–334, 1987.

[Kap88] S. Kaplan. Rewriting with a nondeterministic choice operator. *Theoretical Computer Science*, 56:37–57, 1988.

[KB70] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon, Oxford, U.K., 1970.

[Kir85] C. Kirchner. *Methodes et outils de conception systematique d'algorithmes d'unification dans les theories equationnelles.* PhD thesis, l'Universite de Nancy, 1985.

[Kir87] H. Kirchner. Schematization of infinite sets of rewrite rules: Application to the divergence of completion processes. In *Proceedings of the Second International Conference on Rewriting Techniques and Applications*, pages 180–191, Bordeaux, France, May 1987. Vol. 256 of *Lecture Notes in Computer Science*, Springer, Berlin.

[KJ87] S. Kaplan and J.-P. Jouannaud. *Proceedings of the First International Workshop on Conditional Rewriting.* July 1987. Lecture Notes in Computer Science volume 308.

[KKSdV91] J.R. Kenneway, J.W. Klop, M.R. Sleep, and F.J. de Vries. Transfinite reductions in orthogonal term rewriting systems. In *Proceedings of the 4th International Conference on rewriting techniques and applications*. Springer-Verlag, 1991. Lecture Notes in Computer Science volume 488.

[Klo80] Jan Willem Klop. *Combinatory Reduction Systems*. Centre for Mathematics and Computer Science, Amsterdam, 1980. Mathematical Centre Tracts No. 127.

[Klo91] Jan Willem Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford University Press, Oxford, 1991. To appear.

[KM91] J.W. Klop and A. Middeldorp. Sequentiality in orthogonal term rewriting systems. *Journal of Symbolic Computation*, 12:161–195, 1991.

[KMN88] D Kapur, D.R. Musser, and P. Narendran. Only prime superpositions need be considered in the Knuth-Bendix procedure. *Journal of Symbolic Computation*, 6(1):19–36, August 1988.

[KN86] D. Kapur and P. Narendran. Np-completeness of the set unification and matching problems. In J. Siekmann, editor, *Proceedings of the 8th International Conference on Automated Deduction*, pages 489–495. Springer-Verlag, 1986.

[KNO90] D. Kapur, P. Narendran, and F. Otto. On ground confluence of term rewriting systems. *Information and Computation*, May 1990.

[KO90] S. Kaplan and M. Okada. 2nd international workshop on conditional and typed rewriting systems. June 1990. to appear in Lecture Notes in Computer Science.

[Kru60] J.B. Kruskal. Well-quasi-ordering, the tree theorem, and vazsonyi's conjecture. *Transactions of the American Mathematical Society*, 95:210–225, 1960.

[KSZ90] D. Kapur, G. Sivakumar, and H. Zhang. A new method for proving termination of ac-rewrite systems. In *Proc. of Tenth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 133–148, December 1990. Springer Verlag LNCS 472.

[KZ91] D. Kapur and H. Zhang. A case study of the completion procedure: proving ring commutativity problems. In Lassez and Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 360–394. MIT Press, 1991.

[Lan75] D. Lankford. Canonical algebraic simplification in computational logic. Technical Report Memo ATP-25, Automatic Theorem Proving Project, University of Texas, Austin, Texas, 1975.

[Lan79] D. Lankford. On proving term rewriting systems are noetherian. Technical Report Memo MTP-3, Mathematics Department, Louisiana Tech. University, Ruston, Louisiana, 1979.

[LB77] D. Lankford and A.M. Ballantyne. Decision problems for simple equational theories with commutative-associative axioms: Complete sets of commutative-associative reductions.

Technical Report Memo ATP-39, Department of Mathematics and Computer Science, University of Texas, Austin, TX, 1977.

[LBB84] D. Lankford, G. Butler, and A. Ballantyne. A progresss report on new decision algorithms for finitely presented abelian groups. In *Proceedings of the 7th International Conference on Automated Deduction*, pages 128–141, May 1984. Lecture Notes in Computer Science volume 170.

[Les84] P. Lescanne. Term rewriting systems and algebra. In *Proceedings of the 7th International Conference on Automated Deduction*, pages 166–174, May 1984. Lecture Notes in Computer Science volume 170.

[Les90] P. Lescanne. On the recursive decomposition ordering with lexicographical status and other related orderings. *Journal of Automated Reasoning*, 6:39–49, 1990.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. 2nd edn.

[Lov78] D. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, New York, 1978.

[LP90] S.-J. Lee and D. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 1990. to appear.

[McC89a] W. W. McCune. *Otter 1.0 Users' Guide.* Mathematics and Computer Science Division, Aggonne National Laboratory, Argonne, Illinois, January 1989.

[McC89b] W. W. McCune. Skolem functions and equality. *Association for Automated Reasoning Newsletter*, 12:6–7, May 1989.

[McK75] R. McKenzie. On spectra,and the negative solution of the decision problem for identities having a finite non-trivial model. *Journal of Symbolic Logic*, 40:186–196, 1975.

[Met83] Y. Metivier. About the rewriting systems produced by the Knuth-Bendix completion algorithm. *Information Processing Letters*, 16(1):31–34, January 1983.

[Mid90] A. Middeldorp. *Modular Properties of Term Rewriting Systems.* PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[MN90] Ursula Martin and Tobias Nipkow. Ordered rewriting and confluence. In Mark Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, pages 366–380. Springer-Verlag, 1990.

[Mus80] D.R. Musser. On proving inductive properties of abstract data types. In *Proceedings, 7th ACM Symposium on Principles of Programming Languages*, pages 154–162, January 1980.

[New42] M.H.A. Newman. On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics*, 43(2):223–243, 1942.

[NI89] T. Naoi and Y. Inagaki. Algebraic semantics and complexity of term rewriting systems. In *Proceedings of the 3rd International Conference on rewriting techniques and applications*, pages 311–325, April 1989. Lecture Notes in Computer Science vol. 355.

[NO80] G. Nelson and D. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27:356–364, 1980.

[NRS89] W. Nutt, P. Rety, and G. Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, 7(3 & 4):295–318, 1989. Special issue on unification. Part one.

[O'D85] M.J. O'Donnell. *Equational logic as a programming language.* MIT Press, Cambridge, Mass., 1985.

[Pet83] G.E. Peterson. A technique for establishing completeness results in theorem proving with equality. *SIAM J. on Computing*, 12(1):82–100, February 1983.

[Pet90] Gerald Peterson. Complete sets of reductions with constraints. In Mark Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, pages 381–395. Springer-Verlag, 1990.

[Pla78] D. Plaisted. A recursively defined ordering for proving termination of term rewriting systems. technical report R-78-943, University of Illinois at Urbana-Champaign, Urbana, IL, 1978.

[Pla84] D. Plaisted. An associative path ordering. In *Proceedings of an NSF workshop on the rewrite rule laboratory*, pages 123 – 136, April 1984.

[Pla86] D. Plaisted. A simple non-termination test for the Knuth-Bendix method. In *Proceedings of the Eighth International Conference on Automated Deduction*, pages 79–88, 1986. Lecture Notes in Computer Science 230.

[Pla87] D. Plaisted. A logic for conditional term rewriting systems. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Rewriting*, pages 212 – 227, July 1987.

[Plo72] G. Plotkin. Building-in equational theories. In *Machine Intelligence*, volume 7, pages 73–90. 1972.

[PP91] D. Plaisted and R Potter. Term rewriting: Some experimental results. *Journal of Symbolic Computation*, 11:149 – 180, 1991.

[PS81] G.E. Peterson and M.E. Stickel. Complete sets of reductions for some equational theories. *J. Assoc. Comput. Mach.*, 28(2):233–264, 1981.

[Ŕ87] Pierre Réty. Improving basic narrowing techniques. In *Proceedings of the Second International Conference on Rewriting Techniques and Applications*, pages 228–241, Bordeaux, France, May 1987. Vol. 256 of *Lecture Notes in Computer Science*, Springer, Berlin.

[Red85] U. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the 1985 Symposium on Logic Programming*, pages 138–151, 1985.

[Rob65] J. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12:23–41, 1965.

[SA91] R. Socher-Ambrosius. Boolean algebra admits no convergent term rewriting system. In *Proceedings of the 4th International Conference on rewriting techniques and applications*. Springer-Verlag, 1991. Lecture Notes in Computer Science volume 488.

[Sny89] W. Snyder. Efficient ground completion: an $o(nlogn)$ algorithm for generating reduced sets of ground rewrite rules equivalent to a set of ground equations. In *Proceedings of the 3rd International Conference on rewriting techniques and applications*, pages 419–433, 1989. Lecture Notes in Computer Science, Vol. 355.

[SP88] M. Smith and D. Plaisted. Term rewriting techniques for logic programming i: completion. Technical Report TR88 - 019, University of North Carolina – Chapel Hill, Chapel Hill, NC, 1988.

[SPR89] R.C. Sekar, S. Pawigi, and I.V. Ramakrishnan. Transforming strongly sequential rewrite systems with constructors for efficient parallel execution. In *Proceedings of the 3rd International Conference on rewriting techniques and applications*, pages 404–418, 1989. Lecture Notes in Computer Science, Vol. 355.

[Ste89] J. Steinbach. Extensions and comparison of simplification orderings. In *Proceedings of the 3rd International Conference on rewriting techniques and applications*, pages 434–448, 1989. Lecture Notes in Computer Science, Vol. 355.

[Sti81] M.E. Stickel. A unification algorithm for associative-commutative functions. *Journal of the Association for Computing Machinery*, 28:423–434, 1981.

[SZ90] Joachim Steinbach and Michael Zehnter. Vade-mecum of polynomial orderings. Report SR-90-03, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, West Germany, 1990.

[Tam84] H. Tamaki. Semantics of a logic programming language with a reducibility predicate. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 259–264, February 1984.

[Tis89] S. Tison. Fair termination is decidable for ground systems. In *Proceedings of the 3rd International Conference on rewriting techniques and applications*, pages 462–476, 1989. Lecture Notes in Computer Science, Vol. 355.

[TKB89] Y. Toyama, J.W. Klop, and H.-P. Barendregt. Termination for the direct sum of left-linear term rewriting systems. In *Proceedings of the 3rd International Conference on rewriting techniques and applications*, pages 477–491, 1989. Lecture Notes in Computer Science, Vol. 355.

[Toy87] Y. Toyama. On the church-rosser property for the direct sum of term rewriting systems. *J. Assoc. Comput. Mach.*, 34(1):128–143, January 1987.

[WB83] F. Winkler and B. Buchberger. A criterion for eliminating unnecessary reductions in the Knuth-Bendix algorithm. In *Proceedings of the Colloquium on Algebra, Combinatorics and Logic in Computer Science*, September 1983.

[ZK88] H. Zhang and D. Kapur. First order theorem proving using conditional rewrite rules. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction*, pages 1–20. Springer-Verlag, 1988.

[ZK90] H. Zhang and D. Kapur. Unnecessary inferences in associative-commutative completion procedures. *Mathematical Systems Theory*, 23:175–206, 1990.