A Taste of Rewrite Systems

Nachum Dershowitz

Department of Computer Science, University of Illinois, Urbana IL 61801, USA

Abstract. This survey of the theory and applications of rewriting with equations discusses the existence and uniqueness of normal forms, the Knuth-Bendix completion procedure and its variations, as well as rewriting-based (functional and logic) programming and (equational, first-order, and inductive) theorem proving. Ordinary, associative-commutative, and conditional rewriting are covered. Current areas of research are summarized and an extensive bibliography is provided.

0 Menu

Equational reasoning is an important component in symbolic algebra, automated deduction, high-level programming languages, program verification, and artificial intelligence. Reasoning with equations involves deriving consequences of given equations and finding values for variables that satisfy a given equation.

Rewriting is a very powerful method for dealing with equations. Directed equations, called "rewrite rules", are used to replace equals by equals, but only in the indicated direction. The theory of rewriting centers around the concept of "normal form", an expression that cannot be rewritten any further. Computation consists of rewriting to a normal form; when the normal form is unique, it is taken as the value of the initial expression. When rewriting equal terms always leads to the same normal form, the set of rules is said to be "convergent" and rewriting can be used to check for equality.

This chapter gives a brief survey of the theory of rewriting and its applications to issues in programming languages and automated deduction. Section 1 offers a few motivating examples as appetizer. The "bread and butter" concepts come next: Sect. 2 addresses the question of existence of normal forms; Sect. 3 addresses their uniqueness. Section 4 describes a brew for constructing convergent systems. The main course is a choice of Sect. 5, a method for proving validity of equations, and, Sect. 6, a method for solving equations. As side dishes, Sections 7 and 8 consider extensions of rewriting to handle associative-commutative function symbols and conditional equations. Dessert is Sect. 9, which applies conditional and unconditional rewriting to programming, while, the *pièce de résistance*, Sect. 10, shows how rewriting is used to facilitate deductive and inductive proofs. Finally, over coffee, Sect. 11 mentions some current areas of research. An extensive bibliography for further reading (the "check") is attached.

1 Rewriting

We begin our rewriting feast with the following three puzzles:

Puzzle 1 Hercules and Hydra. Hydra is a bush-like creature with multiple heads. Each time Hercules hacks off a head of hers, Hydra sprouts many new branches identical to—and adjacent to—the weakened branch that used to hold the severed head. But whenever he chops off a head coming straight out of the ground, no new branches result. Suppose Hydra starts off as a lone stalk ten branches tall. The question is: Does Hercules ever defeat Hydra? □

Puzzle 2 Chamelion Island. The chamelions on this strange island come in three colors, red, yellow, and green, and wander about continuously. Whenever two chamelions of different colors meet, they both change to the third color. Suppose there are 15 red chamelions, 14 yellow, and 13 green. Can their haphazard meetings lead to a stable state, all sharing the same color? □

Puzzle 3 Grecian Urn. An urn contains 150 black beans and 75 white. Two beans are removed at a time: if they're the same color, a black one is placed in the urn; if they're different, the white one is returned. The process is repeated as long as possible. Is the color of the last bean in the urn predetermined and, if so, what is it? □

All three puzzles give rules for going from one state to another. The first two concern the possibility of getting from some starting state to a final state; the last asks how the final state is related to the initial state.

In general, a *rewrite system* consists of a set of *rules* for transforming terms. Hercules and Hydra can be expressed pictorially as in Fig. 1.

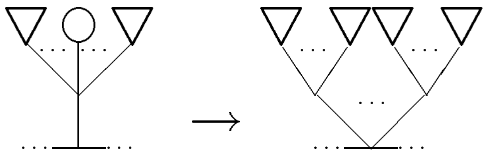


Fig. 1. Hercules versus Hydra.

Chamelion Island can be expressed by three rules:

$$\begin{array}{rcl} \textit{red yellow} & \rightarrow & \textit{green green} \\ \textit{green yellow} & \rightarrow & \textit{red red} \\ \textit{red green} & \rightarrow & \textit{yellow yellow} \ , \end{array}$$

with the added proviso that chamelions can rearrange themselves at will.

The Grecian Urn may be expressed as the following system of four rules, one per possible pair of beans:

$$\begin{array}{rcl} \textit{black black} & \rightarrow & \textit{black} \\ \textit{white white} & \rightarrow & \textit{black} \\ \textit{black white} & \rightarrow & \textit{white} \\ \textit{white black} & \rightarrow & \textit{white} \ . \end{array}$$

In the sequel, we will solve these puzzles by applying techniques from the theory of rewriting. But, first, let's take a look at some more prosaic examples of rewrite systems.

Example 1 (Insertion Sort). The following is a program to arrange a list of numbers in non-increasing order by inserting elements one by one into position:

$$\begin{array}{rcl}
max(0,x) & \rightarrow & x \\
max(x,0) & \rightarrow & x \\
max(s(x),s(y)) & \rightarrow & s(max(x,y)) \\
min(0,x) & \rightarrow & 0 \\
min(x,0) & \rightarrow & 0 \\
min(s(x),s(y)) & \rightarrow & s(min(x,y)) \\
sort(nil) & \rightarrow & nil \\
sort(cons(x,y)) & \rightarrow & insert(x,sort(y)) \\
insert(x,nil) & \rightarrow & cons(x,nil) \\
insert(x,cons(y,z)) & \rightarrow & cons(max(x,y),insert(min(x,y),z)) .
\end{array}$$

Lists are represented in "cons" notation and numbers in successor (unary) notation. We would like to ascertain that every term constructed from $sort$, $cons$, nil, s, and 0 leads to a unique term, not containing $sort$, nor the auxiliary symbols, $insert$, max, and min. □

As in the previous example, both sides of a rule $l \rightarrow r$ can contain variables which refer to arbitrary terms. A rule is used to *rewrite* any subterm that is an instance of the left-hand side l. That is, $u[l\sigma]$ rewrites to $u[r\sigma]$, where σ is a substitution of terms for variables of the rule (the same term substituted for each occurrence of the same variable), $u[\cdot]$ is the "context" in which the instance $l\sigma$ of l occurs as a subterm, and $u[r\sigma]$ is $u[l\sigma]$ with that subterm replaced by the right-hand side after its variables have had the same substitution σ applied. Such a rewrite step is written $u[l\sigma] \rightarrow u[r\sigma]$ or, backwards, as $u[r\sigma] \leftarrow u[l\sigma]$. Rules are applied nondeterministically, since, in general, more than one rule can be applied, and any one rule may apply at more than one position within a term.

Rewrite systems have long been used as decision procedures for validity in equational theories, that is, for truth of an equation in all models of the theory.

Example 2 (Loops). Consider the following system of a dozen rules:

$$\begin{array}{rclcrcl}
x\backslash x & \rightarrow & e & \quad & x \cdot (x\backslash y) & \rightarrow & y \\
x/x & \rightarrow & e & & (y/x) \cdot x & \rightarrow & y \\
e \cdot x & \rightarrow & x & & x\backslash(x \cdot y) & \rightarrow & y \\
x \cdot e & \rightarrow & x & & (y \cdot x)/y & \rightarrow & y \\
e\backslash x & \rightarrow & x & & x/(y\backslash x) & \rightarrow & y \\
x/e & \rightarrow & x & & (x/y)\backslash x & \rightarrow & y .
\end{array}$$

Each rule follows by algebraic manipulation from some combination of the following five axioms for algebraic structures called "loops":

$$\begin{array}{rclcrcl}
x \cdot (x\backslash y) & = & y & \quad & (y/x) \cdot x & = & y \\
x\backslash(x \cdot y) & = & y & & (x/y)\backslash x & = & y
\end{array}$$
$$y/y = x\backslash x ,$$

and the definition $y/y = e$. To use the rewrite system to decide whether an arbitrary equation is a valid identity for all loops (in which case it can be proved by purely equational reasoning), we need to ascertain that *any* two terms equal in the theory have the same normal forms. □

Rewrite systems can also be used to "interpret" other programming languages.

Example 3 (Interpreter). The state of a machine with three integer-valued registers can be represented as a triple $\langle x,\ y,\ z\rangle$. The semantics of its instruction set can be defined by the rules for the interpreter shown in Fig. 2. The penultimate rule, for example, can clearly be applied *ad infinitum*. The question is what strategy of rule application, if any, will lead to a normal form whenever there is one. □

Fig. 2. Three-register machine interpreter.

$$
\begin{array}{rcl}
eval(\mathbf{set0}\ n, \langle x,\ y,\ z\rangle) & \rightarrow & \langle n,\ y,\ z\rangle \\
eval(\mathbf{set1}\ n, \langle x,\ y,\ z\rangle) & \rightarrow & \langle x,\ n,\ z\rangle \\
eval(\mathbf{set2}\ n, \langle x,\ y,\ z\rangle) & \rightarrow & \langle x,\ y,\ n\rangle \\
eval(\mathbf{inc0}, \langle x,\ y,\ z\rangle) & \rightarrow & \langle s(x),\ y,\ z\rangle \\
eval(\mathbf{inc1}, \langle x,\ y,\ z\rangle) & \rightarrow & \langle x,\ s(y),\ z\rangle \\
eval(\mathbf{inc2}, \langle x,\ y,\ z\rangle) & \rightarrow & \langle x,\ y,\ s(z)\rangle \\
eval(\mathbf{dec0}, \langle s(x),\ y,\ z\rangle) & \rightarrow & \langle x,\ y,\ z\rangle \\
eval(\mathbf{dec1}, \langle x,\ s(y),\ z\rangle) & \rightarrow & \langle x,\ y,\ z\rangle \\
eval(\mathbf{dec2}, \langle x,\ y,\ s(z)\rangle) & \rightarrow & \langle x,\ y,\ z\rangle \\
eval(\mathbf{ifpos0}\ p, \langle 0,\ y,\ z\rangle) & \rightarrow & \langle 0,\ y,\ z\rangle \\
eval(\mathbf{ifpos0}\ p, \langle s(x),\ y,\ z\rangle) & \rightarrow & eval(p, \langle s(x),\ y,\ z\rangle) \\
eval(\mathbf{ifpos1}\ p, \langle x,\ 0,\ z\rangle) & \rightarrow & \langle x,\ 0,\ z\rangle \\
eval(\mathbf{ifpos1}\ p, \langle x,\ s(y),\ z\rangle) & \rightarrow & eval(p, \langle x,\ s(y),\ z\rangle) \\
eval(\mathbf{ifpos2}\ p, \langle x,\ y,\ 0\rangle) & \rightarrow & \langle x,\ y,\ 0\rangle \\
eval(\mathbf{ifpos2}\ p, \langle x,\ y,\ s(z)\rangle) & \rightarrow & eval(p, \langle x,\ y,\ s(z)\rangle) \\
\mathbf{whilepos0}\ p & \rightarrow & \mathbf{ifpos0}\ (p;\ \mathbf{whilepos0}\ p) \\
\mathbf{whilepos1}\ p & \rightarrow & \mathbf{ifpos1}\ (p;\ \mathbf{whilepos1}\ p) \\
\mathbf{whilepos2}\ p & \rightarrow & \mathbf{ifpos2}\ (p;\ \mathbf{whilepos2}\ p) \\
eval((p;\ q), u) & \rightarrow & eval(q, eval(p, u))\ .
\end{array}
$$

Two important properties a rewrite system may possess are termination and confluence. We address these in the following two sections.

2 Termination

Definition 1 Termination. A rewrite system is *terminating* if there are no infinite derivations $t_1 \rightarrow t_2 \rightarrow \cdots$.

The rules for the Chamelion Puzzle are not terminating; to wit *red yellow yellow* $\rightarrow$ *green green yellow* $\rightarrow$ *green red red* $\rightarrow$ *yellow yellow red*, which rearranges to *red yellow yellow*, from which point the same three steps may be repeated over and over again. On the other hand, each step in the Grecian Urn Puzzle decreases the number of beans, so it always terminates. The Loop system also terminates, since it always shortens the length of the expression, but, as pointed out above, our Interpreter does not.

Termination is an undecidable property of rewrite systems. Nonetheless, the following general method is very often helpful in termination proofs:

Definition 2 Termination Function. A *termination function* takes a term as argument and is of one of the following types:

- a function that returns the outermost function symbol of a term, with symbols ordered by some precedence (a "precedence" is a well-founded partial ordering of symbols);
- a function that extracts the immediate subterm at a specified position (which position, can depend on the outermost function symbol of the term);
- a function that extracts the immediate subterm of a specified rank (the kth largest in the path ordering defined recursively below);
- a homomorphism from terms to some well-founded set of values;
- a monotonic homomorphism, having the strict subterm property, from terms to some well-founded set (a homomorphism is *monotonic* with respect to the well-founded ordering if the value it assigns to a term $f(\ldots s \ldots)$ is greater than or equivalent to that of $f(\ldots t \ldots)$ whenever the value of s is greater than t; it has the *strict subterm property* if the value of $f(\ldots t \ldots)$ is always strictly greater than that of its subterm t);

– a strictly monotonic homomorphism, having the strict subterm property, from terms to some well-founded set (it is *strictly monotonic* if the value of $f(\ldots s \ldots)$ is strictly greater than that of $f(\ldots t \ldots)$ whenever s is of greater value than t); or
– a constant function.

Simple examples of homomorphisms from terms to the natural numbers are size (number of function symbols, including constants), depth (maximum nesting of function symbols), and weight (sum of integral weights of function symbols). Size and weight are strictly monotonic; depth is monotonic.

Definition 3 Path Ordering. Let $\tau_0, \ldots, \tau_{i-1}$ ($i \geq 0$) be monotonic homomorphisms, all but possibly τ_{i-1} strict, and let $\tau_i, \ldots, \tau_k$ be any other kinds of termination functions. The induced *path ordering* $\succ$ is as follows:

$$s = f(s_1, \ldots, s_m) \succeq g(t_1, \ldots, t_n) = t$$

if either of the following hold:

(1) $s_i \succeq t$ for some s_i, $i = 1, \ldots, m$; or
(2) $s \succ t_1, \ldots, t_n$ and $\langle \tau_1(s), \ldots, \tau_k(s) \rangle$ is lexicographically greater than or equal to $\langle \tau_1(t), \ldots, \tau_k(t) \rangle$, where function symbols are compared according to their precedence, homomorphic images are compared in the corresponding well-founded ordering, and subterms are compared recursively in $\succ$.

A tuple $\langle x_1, \ldots, x_m \rangle$ is *lexicographically* greater than $\langle y_1, \ldots, y_n \rangle$ if (a) x_i is greater than y_i for some i ($1 \leq i \leq n$) and x_j is equivalent to y_j for all j up to, but not including, i, or if (b) the x_i and y_i are equivalent for all i up to and including n and also $m > n$. We say that $s \succ t$ if $s \succeq t$, but $s \not\preceq t$.

This ordering mixes and matches syntactic considerations (the first three types of termination functions) with semantic considerations (the others).

Theorem 4 [19]. *A rewrite system terminates if $l\sigma \succ r\sigma$ in a path ordering $\succ$ for all rules $l \rightarrow r$ and substitutions σ, and also $\tau(l\sigma) = \tau(r\sigma)$ for each of the nonmonotonic homomorphisms among its termination functions.*

The proof of this theorem is based on Kruskal's Tree Theorem and the fact that $s \rightarrow t$ and $s \succ t$ imply $f(\ldots, s, \ldots) \succ f(\ldots, t, \ldots)$, for all terms s, t, ... and function symbols f.

The path ordering encompasses many of the orderings described in the literature. The simplest, and perhaps most generally useful, version of this ordering, called the *lexicographic path ordering*, uses a precedence for τ_0 and the ith subterm for the remaining τ_i (or any other permutation of all the subterms). The precedence typically encapsulates the dependency of some function definitions on others. The resultant path ordering is "syntactic" in that it depends only on the relative order of the function symbols appearing in the terms being compared. Terms s and t are compared by first comparing their outermost function symbols: if they're equal, then subterms are compared recursively, in order; if the outermost symbol of s is larger, all that needs to be shown is that s is larger than t's subterms; otherwise, the only way s can be larger than t is if one of its subterms is.

To prove that Insertion Sort terminates, note that five of the rules show a decrease for $\succ$ by virtue of clause (1) of the definition of the path ordering, regardless of what we choose for the τ_i. For the recursive rules of max and min, we can use the precedence max and min greater than s as τ_0. For the base case of $insert$, we let $insert > cons$ in the precedence. For the recursive rule of $sort$, we let $sort > insert$, and have τ_1 give the sole argument of $sort$. Finally, for the recursive rule of $insert$, we make $insert > max, min$ in the precedence, and have τ_1 return the second argument of $insert$.

Another important class of orderings, the *polynomial path orderings*, uses a polynomial interpretation for τ_0. The interpretation associates a (multivariate) monotonic integer polynomial with each function symbol and constant, which is extended to a homomorphism to give meaning to terms. The remaining τ_i select subterms as before. For example, Loops may be shown terminating by letting each binary symbol take the sum of its arguments. This "semantic" ordering acts like the lexicographic path ordering when the polynomials are all of degree 0 (constants). The advantage of polynomials over other classes of interpretations is that tools are available to help compute inequalities.

To show that Hercules invariably defeats Hydra, we need to show that after each chop and regrowth, Hydra is smaller than before. To this end, we let nodes have variable arity. This *multiset path ordering*, in effect, uses a precedence for τ_0 and has τ_i give the ith largest subterm. The precedence for Hydra makes branching nodes larger than heads. Replacing a branch with any number of smaller branches is a decrease in this ordering.

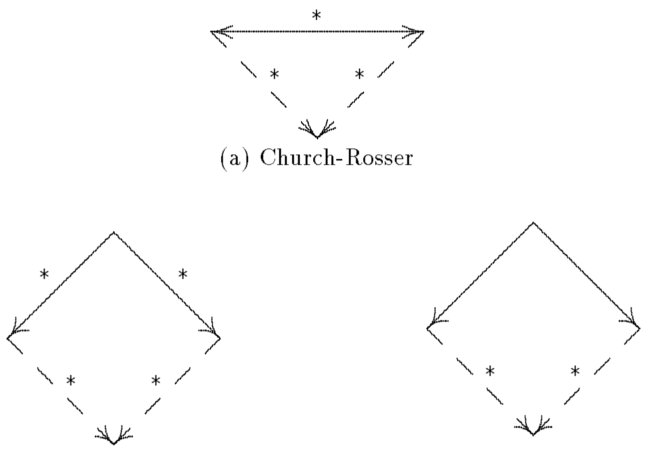
3 Confluence

We will use $s \leftrightarrow t$ to mean $s \rightarrow t$ or $t \rightarrow s$. We say s *derives* t and write $s \rightarrow^* t$ if $s \rightarrow \cdots \rightarrow t$ in zero or more steps. We say that s and t are *convertible*, symbolized $s \leftrightarrow^* t$, if $s \leftrightarrow \cdots \leftrightarrow t$ in zero or more steps, and that s and t are *joinable*, or that $s = t$ has a *rewrite proof*, if s and t derive the same term.

The following two definitions are easily shown equivalent:

Definition 5 Church-Rosser Property. A rewrite system is *Church-Rosser* if terms are joinable whenever they are convertible. (See Fig. 3(a).)

Definition 6 Confluence. A rewrite system is *confluent* if terms are joinable whenever they are derivable from the same term. (See Fig. 3(b).)



(a) Church-Rosser

(b) Globally confluent

(c) Locally confluent

Fig. 3. Confluence Properties.

Terminating confluent systems are called *convergent* or *complete*. Often, we are only interested in confluence for ground (variable-free) terms.

A weaker form of confluence is the following:

Definition 7 Local Confluence. A rewrite system is *locally confluent* if two terms are joinable whenever they are both obtainable by rewriting (in one step) from the same term. (See Fig. 3(c).)

The following, sometimes referred to as the "Diamond Lemma", connects local and global confluence:

Theorem 8 [20]. *A terminating rewrite system is Church-Rosser if and only if it is locally confluent.*

A rewrite system is locally confluent if (but not only if) no left-hand side unifies with a non-variable subterm (except itself) of any left-hand side, taking into account that variables appearing in two rules (or in two instances of the same rule) are always treated as disjoint. To get confluence for nonterminating systems we have an additional requirement:

Definition 9 Orthogonal System. A rewrite system is *orthogonal* if no left-hand side unifies with a renamed non-variable subterm of any other left-hand side or with a renamed proper subterm of itself, and no variable appears more than once on any left-hand side.

The importance of orthogonal systems stems from the following result:

Theorem 10 [22]. *Every orthogonal system is confluent.*

In particular, our erstwhile interpreter is confluent.

Example 4 Combinatory Logic. Combinatory Logic is a prime example of a (nonterminating) orthogonal system (juxtaposition and $\langle\cdot,\cdot\rangle$ are binary operators):

$$\begin{array}{rcl} Ix & \rightarrow & x \\ (Kx)y & \rightarrow & x \\ ((Sx)y)z & \rightarrow & (xz)(yz)\ . \end{array}$$

This system can be used to implement any recursive function. The combinators K and S were dubbed "kestrel" and "starling" by Smullyan; I is the identity combinator. □

Example 5 Cartesian Closed Categories. The following non-orthogonal, non-confluent system is used in the compilation of functional languages (juxtaposition represents the binary composition operator):

$$\begin{array}{rclcrcl} Ix & \rightarrow & x & \qquad & (xy)z & \rightarrow & x(yz) \\ xI & \rightarrow & x & & \langle x,y\rangle z & \rightarrow & \langle xz, yz\rangle \\ F\langle x,y\rangle & \rightarrow & x & & E\langle Cx, y\rangle & \rightarrow & x\langle I, y\rangle \\ S\langle x,y\rangle & \rightarrow & y & & (Cx)y & \rightarrow & C(x\langle yF, S\rangle)\ . \end{array}$$

The combinators E and C stand for "evaluation" and "Currying", respectively; I is the identity morphism; F and S project the components of pairs $\langle\cdot,\cdot\rangle$. □

For terminating confluent systems, any rewriting strategy will lead to the unique normal form of any given term; for nonterminating orthogonal systems, we defer to the discussion in Sect. 9. Also, for terminating systems, there is a stronger result, relating confluence to joinability in a finite number of cases:

Definition 11 Critical Pair. If $l \rightarrow r$ and $s \rightarrow t$ are two rewrite rules (with variables made distinct) and μ is a most general unifier of l and a nonvariable subterm s' of s, then the equation $t\mu = s\mu[r\mu]$, where $r\mu$ has replaced $s'\mu$ $(= l\mu)$ in $s\mu$, is a *critical pair*.

A finite rewrite system has a finite number of critical pairs.

Theorem 12 [28][25]. *A rewrite system is locally confluent if and only if all its critical pairs are joinable. Therefore, a terminating system is confluent (hence, convergent) if and only if all its critical pairs are joinable.*

Insertion Sort has one trivially joinable critical pair, $0 = 0$, formed from either the first two rules, or the fourth and fifth.

Example 6 Fragment of Group Theory. Each of the rules

$$\begin{array}{rcl@{\qquad}rcl} 0+x & \rightarrow & x & x+0 & \rightarrow & x \\ (-x)+x & \rightarrow & 0 & x+(-x) & \rightarrow & 0 \\ -0 & \rightarrow & 0 & -(-x) & \rightarrow & x \\ (-x)+(x+y) & \rightarrow & y & x+((-x)+y) & \rightarrow & y \end{array}$$

follows from some combination of the following three axioms:

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

This system has numerous critical pairs, all of which are joinable. For example, the rules $x+(-x) \rightarrow 0$ and $x+((-x)+y) \rightarrow y$ form a critical pair $x+0 = -(-x)$, both sides of which reduce, via other rules, to x. □

Definition 13 Encompassment. A term s *encompasses* a term t if a subterm of s is an instance of t. We write $s \rhd t$ if s encompasses t, but not vice-versa.

Definition 14 Reduced System. A system R is *reduced* if, for each rule $l \rightarrow r$ in R, the right-hand side r is irreducible (unrewritable) and if $l \rhd l'$, for the left-hand side l, then l' is irreducible (proper subterms of l are in normal form, as is any term more general than l).

The above Fragment is reduced, but the Interpreter isn't.

Definition 15 Canonical System. A rewrite system is *canonical* if it is confluent, terminating, and reduced.

The following interesting fact was observed by a number of people:

Theorem 16 [29]. *Suppose two canonical (not necessarily finite) rewrite systems have the same equational theory (that is, their convertibility relations are the same) and when combined are still terminating. Then the two must be identical up to renaming of variables.*

Two interesting results with practical repercussions are:

Theorem 17 [27]. *The union of two confluent rewrite systems sharing no function symbols (or constants) is also confluent.*

Theorem 18 [94]. *The union of two convergent rewrite systems sharing no function symbols (or constants), and in which no variable appears more than once on any left-hand side, is also convergent.*

4 Completion

To construct confluent systems, a method is used, called *completion*, which turns critical pairs into rewrite rules. Completion uses an ordering to orient equations and to allow use of unoriented equations to simplify. This ordering should be a *reduction ordering*, defined as follows:

Definition 19 Reduction Ordering. A well-founded partial ordering $\succ$ on terms is a *reduction ordering* if $s \succ t$ implies $u[s\sigma] \succ u[t\sigma]$, for any context $u[\cdot]$ and substitution σ.

The path ordering is not necessarily a reduction ordering, but its major special cases, including the lexicographic and multiset path orderings, are.

The encompassment ordering also plays a role in most versions of completion. It is used to determine which of two rules is more general, and hence preferred.

In our version of completion, not only rules, but also equations, are used to rewrite with. A term $u[l\sigma]$ containing an instance of l of an equation $l=r$ or $r=l$ may be rewritten to $u[r\sigma]$ whenever $u[l\sigma]$ is greater than $u[r\sigma]$ in the given reduction ordering.

We require a broader notion of critical pair:

Definition 20 Ordered Critical Pair. Given a reduction ordering $\succ$, if $l = r$ and $s = t$ are two (not necessarily distinct) equations (with variables distinct), μ is a most general unifier of l and a nonvariable subterm of s, and $s\mu$ is not necessarily smaller than either $t\mu$ or $s\mu[r\mu]$, then the equation $t\mu = s\mu[r\mu]$ is a *(ordered) critical pair* formed from those equations. (Term s is not necessarily smaller than term t if there is a substitution σ such that $s\sigma \succ t\sigma$.)

Completion maintains a set of unoriented equations E and a set of rules R oriented according to the given reduction ordering $\succ$. These sets are manipulated in the following six ways:

Deduce: Add a critical pair of R and/or E to E.

Simplify: Use a rule in R to rewrite either side of an equation $s=t$ in E. Or use an equation $l=r$ in E (or use $r=l$ in E) to rewrite s (or t), provided l strictly encompasses s (or t).

Delete: Remove an equation from E whose sides are identical.

Orient: Remove an equation $s=t$ (or $t=s$) from E such that $s \succ t$. Add the rule $s \rightarrow t$ to R.

Compose: Use R or E to rewrite the right-hand side of an existing rule.

Collapse: Use a rule $l \rightarrow r$ in R or equation $l=r$ in E to rewrite the left-hand side s of a rule in R, provided l strictly encompasses s. Remove the rewritten rule from R and place it as an equation in E.

One simple version of completion mixes the above inference steps according to the following strategy:

$$(((\textbf{Simplify} \cup \textbf{Delete})^*;\ (\textbf{Orient};\ \textbf{Compose}^*;\ \textbf{Collapse}^*))^*;\ \textbf{Deduce})^* .$$

In words: simplify and delete equations as much as possible before orienting. Use the newly oriented equation to fully compose right-hand sides and collapse left-hand sides of all non-reduced existing rules; then, go back and simplify over again. When there are no equations left to orient, generate one new critical pair, and repeat the whole process.

Different versions of completion differ in which equations they orient first and in how they keep track of critical pairs that still need to be deduced. They, in fact, often skip some critical pairs. A version of completion is *fair* if it does not altogether avoid processing any relevant critical pair. Running completion with a fair strategy can have one of three outcomes: It might converge on a finite system that is a decision procedure for the initial set of equations; it might reach a point in which all (ordered) critical pairs have been considered and all have rewrite proofs—using rules *and* equations; or it might loop and generate an infinite number of rules and/or equations.

Theorem 21 [43][40]. *For any fair completion strategy, if at some point all critical pairs between persisting rules have been considered and no equation persists forever in E, then the (finite or infinite) set of rules persisting in R is convergent.*

This means that eventually both sides of any identity in the theory of the initial set of equations will become joinable.

Example 7 Abelian Groups I. Completion, given

$$\begin{array}{rcl@{\qquad}rcl} x \cdot 1 &=& x & x \cdot y &=& y \cdot x \\ x \cdot (y \cdot z) &=& (x \cdot y) \cdot z & x \cdot x^- &=& 1\ , \end{array}$$

and a lexicographic path ordering (in which $^- > \cdot > 1$ and $\cdot$ looks at its left argument first) will generate the following decision procedure for free Abelian (commutative) groups:

$$\begin{array}{rcl@{\qquad}rcl} 1^- &\to& 1 & x \cdot y &\leftrightarrow& y \cdot x \\ x \cdot 1 &\to& x & x \cdot (y \cdot z) &\leftrightarrow& y \cdot (x \cdot z) \\ 1 \cdot x &\to& x & (x \cdot y) \cdot z &\to& x \cdot (y \cdot z) \\ (x^-)^- &\to& x & x \cdot (x^- \cdot z) &\to& z \\ x \cdot x^- &\to& 1 & (y \cdot x)^- &\to& x^- \cdot y^-\ . \end{array}$$

Those equations used in both directions have a two-headed arrow. To decide validity of an equation $s = t$, the lexicographic path ordering is extended to a total ordering that includes any constants appearing in s or t. The double-headed rules are then used only in a direction that reduces in this ordering. The equation is valid if and only if both s and t have the same normal form. □

Definition 22 Completable Simplification Ordering. A reduction ordering is a *completable simplification ordering* if it can be extended to a reduction ordering that is total on ground terms (in which any two distinct terms—made up of function symbols and constants occurring in the given equations—are comparable).

Such orderings have the property that every term is strictly greater than its proper subterms. Examples include the empty ordering, the lexicographic path ordering with a partial or total precedence, and the polynomial path ordering.

Theorem 23 [33][40]. *Suppose R is a finite convergent system for axioms E and $\succ$ is a completable simplification ordering for which all rules in R decrease. Any fair completion strategy will generate a finite convergent system for E (not necessarily identical to R).*

If R is canonical and the strategy performs all compositions and collapses, then, by Theorem 16, completion will actually produce R.

The efficiency of completion depends on the number of critical pairs deduced. In general, an equation $s = t$ is *redundant* if there exist proofs Q_i of equations $u_{i-1} = u_i$, $i = 1, \ldots, n$, such that $u_0 = s$, $u_{n+1} = t$, $u \succ u_0, \ldots, u_{n+1}$, and the proof $s \leftarrow u \rightarrow t$ from which the critical pair was formed is greater than each of the Q_i in some well-founded ordering $\gg$ on equational proofs. (The proof ordering $\gg$ must have the property that a proof decreases by replacing a subproof with one involving terms that are all smaller vis-a-vis the reduction ordering $\succ$ supplied to completion.)

The following can be shown by induction with respect to $\gg$:

Theorem 24 [40]. *For any fair completion strategy, if at some point all non-redundant critical pairs between persisting rules have been considered and no equation persists forever in E, then the (finite or infinite) set of rules persisting in R is convergent.*

For the Group Fragment, the critical pair $y = (-0) + y$, obtained by rewriting $(-0) + (0 + y)$ with $0 + x \rightarrow x$ and $(-x) + (x + y) \rightarrow y$, is redundant since $(-0) + (0 + y)$ is greater than each of the terms in the alternative proof $y \leftarrow 0 + y \leftarrow (-0) + y$.

Various techniques have been used in practice to check for redundancy. In particular, a critical pair can be ignored if the variable part of either of the rules involved becomes reducible. Experimental results that give some indication of the utility of such redundancy criteria have been reported, particularly in the associative-commutative case described in the next section.

5 Validity

In addition to its use for generating canonical systems to serve as decision procedures for the given axioms, completion may also be used as an equational theorem prover. Classical forward reasoning systems work from the axioms, "expanding" the set of established formulæ by inferring new ones from old ones. Completion may be viewed as an inference engine that also "contracts" formulæ by constantly rewriting them, making forward reasoning practical. The potentially infinite set of rules and equations generated by completion are used to simplify the two sides of the equation in question. Rules are used in the indicated direction only, while equations are used in whichever direction reduces the term it is applied to in the given ordering. An identity is proved when both sides reduce via rules and equations to the identical term.

For completeness of this theorem-proving methodology, we require a completable ordering.

Theorem 25 [75][44][33]. *Suppose $s=t$ is a theorem of E. For any fair completion strategy starting with E and a completable simplification ordering, at some point s and t will rewrite to the identical term using the generated rules and equations.*

The word problem in arbitrary equational theories can always be semi-decided in this way. With an empty ordering, completion amounts to ordinary paramodulation of unit equations; with more of an ordering, completion can be more effective, by reducing the number of allowed inferences and increasing the amount of simplification that may be performed without loss of completeness.

Example 8 Distributive Lattices. With axioms:

$$\begin{array}{rcl@{\qquad}rcl}
x \wedge x & = & x & x \vee x & = & x \\
x \wedge y & = & y \wedge x & x \vee y & = & y \vee x \\
(x \wedge y) \wedge z & = & x \wedge (y \wedge z) & (x \vee y) \vee z & = & x \vee (y \vee z) \\
(x \vee y) \wedge x & = & x & (x \wedge y) \vee x & = & x
\end{array}$$

$$x \vee (y \wedge z) \quad = \quad (x \wedge y) \vee (x \wedge y) \; ,$$

and a lexicographic path ordering that makes meet bigger than join, completion eventually generates:

$$x \wedge (y \vee z) \quad \rightarrow \quad (x \vee y) \wedge (x \vee y) \; .$$

□

The above method can be refined to ignore critical pairs that have different rewrite proofs, each depending on the relative ordering of terms substituted for variables.

6 Satisfiability

We turn now from questions of validity to satisfiability.

If ground terms are joinable whenever they are convertible, we say the system is *ground confluent.* If it is, in addition, terminating, then it's *ground convergent*:

Definition 26 Ground Convergence. A system is *ground convergent* if every ground (variable-free) term always rewrites to a unique normal form.

Ground convergence is a reasonable property to expect from a rewrite system, like Insertion Sort, used as a program. Such programs compute the value of a ground term by rewriting it to its unique normal form.

Definition 27 Narrowing. A term s *narrows* to a term t, symbolized $s \rightsquigarrow t$, if $t = s\mu[r\mu]$, for some non-variable subterm s' of s, renamed rule $l \rightarrow r$ in R, and most general unifier μ of s' and l.

By $\rightsquigarrow^*$ we denote the reflexive-transitive closure of this narrowing relation.

Theorem 28 [48]. *If R is a rewrite system, σ is an irreducible substitution (that is, $x\sigma$ is irreducible for all variables x), and $s\sigma \rightarrow^* t$, then there exists a term u such that $s \rightsquigarrow^* u$ and u is at least as general as t.*

A term u is at least as general as a term t if there is a substitution τ such that $t\tau = u$.

Thus, narrowing can be used with such systems to solve equational goals; a solution is a substitution σ for which $s\sigma$ has normal form t. Narrowing a goal $eval(p, \langle x, 0, 0\rangle)$ with the Interpreter, where p is some fixed program, will find all inputs x that lead to various final triples. The unifiers of each step are composed to get the solution to the original goal.

Theorem 29 [8]. *If R is a ground convergent rewrite system and $s\sigma \rightarrow^* t$, then there exist terms u and v such that $s \rightsquigarrow^* u$, $t \rightarrow^* v$, and u is at least as general as v.*

Example 9 List Append. With the program

$$\begin{array}{rcl} append(nil, y) & \rightarrow & y \\ append(cons(x, z), y) & \rightarrow & cons(x, append(z, y)) \end{array}$$

the goal $append(x, y)$ generates all lists x and the result of appending them to y. For example, $append(x, y) \rightsquigarrow y$ with $x \mapsto nil$. □

Without convergence, reducible solutions are lost. Variations on narrowing include: *normal narrowing* in which terms are rewritten to normal form before narrowing; *basic narrowing* in which the substitution part of prior narrowings is not subsequently narrowed; *top-down narrowing* in which terms are decomposed from top down to identify necessary narrowings; *left-to-right narrowing* in which the leftmost possible narrowing is performed at each step; other restrictions obtain all solutions only in special cases.

7 Associativity and Commutativity

Many axioms are difficult to handle by rewriting. We could not, for example, include the non-terminating rule $x + y \rightarrow y + x$ in a rewrite-system decision procedure. Instead, we can restrict application of the rule to commute only in the direction that decreases the term in some given ordering. For example, we might allow $2+1$ to be replaced by $1+2$, but not vice-versa, as done in Sect. 5. Another approach is to use commutativity only to enable the application of other rules. For example, we would apply a rule $x + 0 \rightarrow x$ to $0 + 1$, as well as to $1 + 0$. In the associative-commutative (AC) case, this means that $u[s] \rightarrow u[r\sigma]$ whenever s is equal under AC to an instance $l\sigma$ of the left-hand side of some rule $l \rightarrow r$ (that is, s and $l\sigma$ may have arguments to AC symbols permuted). Thus, AC-matching must be used to detect applicability of rules in this case for AC-rewriting.

To better handle these problematic identities, reasonably efficient special-purpose completion procedures have been designed. For instance, the completion procedure given in the previous section may be modified in the following ways to handle AC symbols:

1. An equation $s = t$ is oriented only if $s' \succ t'$ for any AC variants s' of s and t' of t. This ensures that each AC-rewrite reduces the term it is applied to.
2. AC-unification is used to generate critical pairs instead of ordinary unification. This means that we look at the *set* of most general substitutions that allow a left-hand side to be AC-equal to a nonvariable subterm of another left-hand side, and get a critical pair for each such ambiguously rewritable term.
3. AC-rewriting is used for composition, collapsing, and simplification.
4. Any equation between AC-variants is deleted.
5. An additional expansion operation is needed: whenever a new equation $f(s, t) = r$ is formed, where f is an AC symbol and $f(s, t) \not\succ r$, an *extended* equation $f(s, f(t, z)) = f(r, z)$ is added, too, for some new variable z. This ensures that $f(s, t) = r$ can be used even when rearrangement is needed to get an instance of its left-hand side.

Recall that the Chamelions do not terminate. But if we turn any of the rules around, they do! Though the result is not confluent, AC completion can be used to generate a confluent system of rules and equations: Start off with the three rules as unoriented equations, and use an ordering with *red* $>$ *yellow* $>$ *green*. The oriented equations are:

$$\begin{array}{rcl} \mathit{red}\ \mathit{yellow} & \rightarrow & \mathit{green}\ \mathit{green} \\ \mathit{red}\ \mathit{red} & \rightarrow & \mathit{green}\ \mathit{yellow} \\ \mathit{red}\ \mathit{green} & \rightarrow & \mathit{yellow}\ \mathit{yellow}\ . \end{array}$$

Notice how the second rule acts contrary to the "real" chamelions. The extended rules are:

$$\begin{array}{rcl} \mathit{yellow}\ (\mathit{red}\ z) & \rightarrow & \mathit{green}\ (\mathit{green}\ z) \\ \mathit{red}\ (\mathit{red}\ z) & \rightarrow & \mathit{green}\ (\mathit{yellow}\ z) \\ \mathit{green}\ (\mathit{red}\ z) & \rightarrow & \mathit{yellow}\ (\mathit{yellow}\ z)\ , \end{array}$$

and their commutative variants. The *green red* and extended *yellow red* rules produce an critical pair $\mathit{yellow}\ (\mathit{yellow}\ \mathit{yellow}) = \mathit{green}\ (\mathit{green}\ \mathit{green})$, which gets oriented from left to right:

$$\mathit{yellow}\ (\mathit{yellow}\ \mathit{yellow}) \quad \rightarrow \quad \mathit{green}\ (\mathit{green}\ \mathit{green})\ .$$

The complete system reduces the initial state and the monochrome states to distinct normal forms. (Which?) Since the system is Church-Rosser, there is no way to get from the initial arrangement of chamelions to one in which they are of uniform color, no matter which way any of the rules are used.

Example 10 Abelian Groups II. The AC-completion procedure, given

$$\begin{array}{rcl} x \cdot 1 & = & x \\ x \cdot x^- & = & 1\ , \end{array}$$

where $\cdot$ is AC, and a polynomial ordering in which $\tau_0(x \cdot y) = \tau_0(x) + \tau_0(y) + 1$, $\tau_0(x^-) = 2\tau_0(x)$, and $\tau_0(1) = 1$, generates the following decision procedure for Abelian groups:

$$\begin{array}{rclcrcl} 1^- & \rightarrow & 1 & \qquad & x \cdot 1 & \rightarrow & x \\ (x^-)^- & \rightarrow & x & & (y \cdot x)^- & \rightarrow & x^- \cdot y^- \\ x \cdot x^- & \rightarrow & 1 & & x \cdot (x^- \cdot z) & \rightarrow & z\ . \end{array}$$

The last rule is a composed version of the extension $x \cdot (x^- \cdot z) \rightarrow 1 \cdot z$ of $x \cdot x^- \rightarrow 1$. This extended rule, together with with $(y_0 \cdot x_0)^- \rightarrow x_0^- \cdot y_0^-$, forms a critical pair

$$(x_2 \cdot z_1)^-(x_2 \cdot (x_1 \cdot x_2)^- \cdot z_2)^- \quad = \quad (z_1 \cdot z_2)^-$$

via AC-unifier $x \mapsto x_1 \cdot x_2$, $z \mapsto z_1 \cdot z_2$, $y_0 \mapsto x_1 \cdot z_1$, and $x_0 \mapsto x_2 \cdot x^- \cdot z_2$. Both sides reduce to $z_2^- \cdot z_1^-$.

With this system, both sides of any identity reduce by AC-rewriting to AC-equal terms. □

The Grecian Urn may also be viewed as an AC-rewriting system. Beans are rearranged until the pair to be removed are adjacent. Extended with rules $l\ z \rightarrow r\ z$ for each original bean rule $; \rightarrow r$, this is system is confluent. For instance, *black white* $\rightarrow$ *white* and the extended rule *white white z* $\rightarrow$ *black z*, rewrite a *black white white* bean arrangement to *white white* and *black black*, respectively, both of which rewrite to *black*. This means that the normal form is independent of the order in which rules are applied. If there are an even number of white beans, they can be paired and reduced to black, and then all the black beans reduce to one; if there are an odd number, the leftover white bean swallows all remaining blacks.

Example 11 Ring Idempotents. The ring axioms

$$\begin{array}{rcl@{\qquad\qquad}rcl} x+0 &=& x & x(y+z) &=& (xy)+(xz) \\ x+(-x) &=& 0 & (y+z)x &=& (yx)+(zx) \\ x+y &=& y+x & (x+y)+z &=& x+(y+z) \\ (xy)z &=& x(yz)\ , & & & \end{array}$$

plus

$$\begin{array}{c} aa=a \qquad\qquad bb=b \qquad\qquad cc=c \\ (a+b+c)(a+b+c) \quad = \quad a+b+c\ , \end{array}$$

can be completed, with an appropriate ordering, to a convergent AC system that includes the rules

$$\begin{array}{rcl} ba &\rightarrow& -(ab)+-(bc)+-(cb)+-(ac)+-(ca) \\ bca &\rightarrow& abc+acb+cab+cac+cbc+ab+ab+ac+ac+cb+cb+bc+ca\ . \end{array}$$

The normal forms of this system include (besides inverses and sums) all monomials (the order of factors matters) not containing aa, bb, cc, ba, or bca. □

8 Conditional Rewriting

A *conditional rewrite system* is a collection of rules of the form

$$u_1 = v_1 \ \wedge \cdots \wedge \ u_n = v_n \ \mid \ l \quad \rightarrow \quad r\ ,$$

meaning that terms $u[l\sigma]$, containing an instance of left-hand side l, rewrite to $u[r\sigma]$ only when all the conditions $u_i\sigma = v_i\sigma$ hold. The most popular operational semantics for such a system require both sides of each condition to rewrite to the same normal form before an instance of l may be rewritten.

Example 12 Conditional Append. To get the flavor of this expanded notion of rewriting, consider the system

$$\begin{array}{rrcl} & null(nil) &\rightarrow& true \\ & null(cons(x,y)) &\rightarrow& false \\ & car(cons(x,y)) &\rightarrow& x \\ & cdr(cons(x,y)) &\rightarrow& y \\ & append(nil,y) &\rightarrow& y \\ null(x)=false \ \mid & append(x,y) &\rightarrow& cons(car(x), append(cdr(x),y)) \end{array}$$

and its derivation

$$append(cons(a, cons(b, nil)), cons(c, nil)) \xrightarrow{*} cons(a, cons(b, cons(c, nil)))\ .$$

□

Definition 30. A conditional rewrite system is *orthogonal* if each variable occurs at most once in a left-hand side of each rule, one side of each condition in each rule is a ground normal form, and no left-hand side unifies with a renamed non-variable subterm of any other left-hand side or with a proper subterm of itself.

Theorem 31 [56]. *Every orthogonal conditional rewrite system is confluent.*

This definition of orthogonality could be weakened to allow overlaps when the conjunction of the conditions of the overlapping rules cannot be satisfied by the rules of the system. This is the case with the Conditional Append example, since only the last two rules overlap, but $null(nil)$ can never be $false$.

For non-orthogonal systems, another approach is required. Under certain circumstances terminating systems are confluent if all their critical pairs are joinable, but we need a suitable notion of critical pair:

Definition 32 Conditional Critical Pair. Let R be a conditional rewrite system. The conditional equation $c\mu \wedge p\mu \Rightarrow s\mu[r\mu] = t\mu$ is a *conditional critical pair* of R if $c \mid l \rightarrow r$ and $p \mid s \rightarrow t$ are conditional rules of R, l unifies via most general unifier μ with a nonvariable subterm of s and $c\mu \wedge p\mu$ is satisfiable in R.

Theorem 33 [59]. *If no left-hand side unifies with a nonvariable* proper *subterm of a left-hand side, and if every critical pair is joinable, then the system is confluent.*

A conditional critical pair $p \Rightarrow s = t$ is joinable if $s\sigma$ and $t\sigma$ are joinable for all σ satisfying p.

Definition 34 Decreasing Systems. A conditional system is *decreasing* if there exists a well-founded ordering $\succ$ containing the rewrite relation $\rightarrow$ and which satisfies two additional requirements: (a) $\succ$ has the subterm property $f(\ldots, s, \ldots) \succ s$, and (b) $l\sigma \succ c\sigma$ for each rule $c \mid l \rightarrow r$ and substitution σ.

Decreasing systems exactly capture the finiteness of recursive evaluation of terms. The notion needs to be extended, however, to cover systems (important in logic programming) with variables in conditions that do not also appear in the left-hand side.

Theorem 35 [57][59]. *A decreasing system is confluent (hence, convergent) if and only if there is a rewrite proof of $s\sigma = t\sigma$ for each critical pair $c \Rightarrow s = t$ and substitution σ such that $c\sigma$ holds.*

Conditional Append is decreasing.

Example 13 Stack. The following is a decreasing conditional rewrite system:

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

□

Completion has been extended to conditional equations, with "ordered" conditional critical pairs turned into rules only if they are decreasing.

9 Programming

Rewrite systems are readily used as a programming language. If one requires of the programmer that all programs be terminating, then rewriting may be used as is to compute normal forms. With ground confluence, one is assured of their uniqueness.

Many programs (interpreters, for example) do not always terminate. Still, we would want to compute normal forms whenever they exist. Confluent systems have at most one normal form per input term. To find the unique normal form for orthogonal systems, we use the following strategy for choosing the point at which to apply a rule:

Definition 36 Outermost Rewriting. A rewriting step $s \rightarrow t$ is *outermost* if no rule applies at a symbol closer to the root symbol (in the tree representation of terms).

Theorem 37 [64]. *For any orthogonal system, if no outermost step is perpetually ignored, the normal form—if there is one—will be reached.*

Outermost rewriting of expressions is used to compute normal forms in Combinatory Logic.

In this way, orthogonal systems provide a simple, pattern-directed (first-order) functional programming language, in which the orthogonal conditional operator

$$\begin{array}{rcl} if(true, x, y) & \rightarrow & x \\ if(false, x, y) & \rightarrow & y \end{array}$$

can also conveniently be incorporated. Various strategies have been developed for efficient computation in special cases. Moreover, orthogonal systems lend themselves easily to parallel evaluation schemes.

Conditional equations provide a natural bridge between functional programming, based on equational semantics, and logic-programming, based on Horn clauses. Interpreting definite Horn clauses $p \vee \neg q_1 \vee \ldots \vee \neg q_n$ as conditional rewrite rules, $q_1 = true \wedge \cdots \wedge q_n = true \mid p \rightarrow true$ gives a system satisfying the constraints of Theorem 33, because predicate symbols are never nested in the "head" (p) of a clause. Furthermore, all critical pairs are joinable, since all right-hand sides are the same ($true$).

Solving existential queries for conditional equations corresponds to the logic-programming capability of resolution-based languages like Prolog. Goals of the form $s \rightarrow^? t$ can be solved by a linear restriction of paramodulation akin to narrowing (for unconditional equations) and to the SLD-strategy in Horn-clause logic. If s and t are unifiable, then the goal is satisfied by any instance of their most general unifier. Alternatively, if there is a (renamed) conditional rule $c \mid l \rightarrow r$ such that l unifies with a nonvariable (selected) subterm of s via most general unifier μ, then the conditions in $c\mu$ are solved, say via substitution ρ, and the new goal becomes $s\mu\rho \rightarrow^? t\mu\rho$.

Suppose we wish to solve

$$append(x, y) \quad \stackrel{?}{\rightarrow} \quad x$$

using Conditional Append. To apply the conditional rule, we need first to solve $null(x) =^? false$ using the (renamed) rule $null(cons(u, v)) \rightarrow false$, thereby narrowing the original goal to

$$cons(car(cons(u, v)), append(cdr(cons(u, v)), y)) \quad \stackrel{?}{\rightarrow} \quad cons(u, v)\ .$$

Straightforward rewriting reduces this to

$$cons(u, append(v, y)) \quad \stackrel{?}{\rightarrow} \quad cons(u, v)\ ,$$

to which the first rule for $append$ applies (letting v be nil), giving a new goal $cons(u, y) \rightarrow^? cons(u, v)$. Since the two terms are now unifiable, this process has produced the solution $x \mapsto cons(u, nil)$ and $y, v \mapsto nil$.

For ground confluent conditional systems, any equationally satisfiable goal can be solved by the method outlined above. Some recent proposals for logic programming languages, incorporating equality, adopt such an operational mechanism.

Simplification via terminating rules is a very powerful feature, particularly when defined function symbols are allowed to be arbitrarily nested in left-hand sides (which is not permitted with orthogonal rules). Assuming ground convergence, any strategy can be used for simplification, and completeness of the goal-solving process is preserved. One way negation can be handled is by incorporating negative information in the form of rewrite rules which are then used to simplify subgoals to $false$. Combined with eager simplification, this approach has the advantage of allowing unsatisfiable goals to be pruned, thereby avoiding some potentially infinite paths. Various techniques are also available to help avoid some superfluous paths that cannot lead to solutions.

10 Theorem Proving

We have already seen how completion is used to prove validity of equations in equational theories (that is, truth in all models) and also how narrowing and related methods can be used to solve equational goals in conditional and unconditional theories. In this section we will see how to handle the non-equational case.

Completion can be used to prove validity of Horn clauses in Horn theories by writing each clause $\neg q \vee p$ as an equation $p \wedge q = q$, and each unit clause p as $p = \mathit{true}$. The completeness of completion for equational reasoning means that an equational proof will be found for any atomic formula, the validity of which follows from a given set of clauses, all expressed as just indicated (plus the Boolean equation $\mathit{true} \wedge x = x$).

One approach to extending rewriting methods to the full first-order predicate calculus is to add inference rules for restricted forms of paramodulation to refutational theorem provers. An ordering of terms and formulæ is used to restrict inferences and to allow simplification.

Alternatively, one can apply completion to the full first-order case by employing Boolean algebra. The following AC system does the trick:

$$
\begin{array}{rcl@{\qquad}rcl}
x \vee \mathit{true} & \rightarrow & \mathit{true} & x \equiv \mathit{true} & \rightarrow & x \\
x \vee \mathit{false} & \rightarrow & x & (x \equiv y) \vee z & \rightarrow & (x \vee z) \equiv (y \vee z) \\
x \vee y & \leftrightarrow & y \vee x & x \vee (y \vee z) & \leftrightarrow & (x \vee y) \vee z \\
x \equiv y & \leftrightarrow & y \equiv x & x \equiv (y \equiv z) & \leftrightarrow & (x \equiv y) \equiv z \\
x \vee x & \rightarrow & x & x \vee x \vee y & \rightarrow & x \vee y \\
x \equiv x & \rightarrow & \mathit{true} & x \equiv x \equiv y & \rightarrow & y \ ,
\end{array}
$$

where $\vee$ is "inclusive or" and $\equiv$ is "equivalent". With this system, all propositional tautologies reduce to *true* and contradictions, to *false*. To prove validity of a formula, one Skolemizes its negation (and renames variables) to obtain a universally quantified formula, and expresses it using the above connectives. Then, were the original formula true, there would be an equational proof of the contradiction $\mathit{true} = \mathit{false}$. Completion can be used to discover such a proof.

As a very simple example, consider the theorem

$$[\exists x p(x) \wedge \forall x\, (p(x) \supset p(f(x)))] \ \supset\ \exists x p(f(f(x))) \ .$$

Its Skolemized negation is equivalent to the following equations:

$$
\begin{array}{rcl}
p(f(f(x))) & = & \mathit{false} \\
p(a) & = & \mathit{true} \\
p(x) \vee p(f(x)) & = & p(f(x)) \ ,
\end{array}
$$

where a is a Skolem constant. These equations entail the contradiction:

$$
\begin{array}{rcl}
\mathit{true} = \mathit{true} \vee [p(f(a)) \vee p(f(f(a)))] = & p(a) \vee [p(f(a)) \vee p(f(f(a)))] & = \\
{[p(a) \vee p(f(a))] \vee p(f(f(a)))} \ = & p(f(a)) \vee p(f(f(a))) \ = & p(f(f(a))) = \mathit{false} \ .
\end{array}
$$

For program verification, one often needs to prove that an equation or formula holds in the standard (initial) model, rather than in all models. Such proofs typically require the use of induction. Rewriting techniques have been applied to inductive proofs of equations, establishing $s = t$ in the standard model of the axioms by proving that $s\sigma$ and $t\sigma$ are joinable for all substitutions σ of ground terms constructed from function symbols and constants appearing in the axioms. For example, the equation $\mathit{append}(x, \mathit{nil}) = x$ is true for all lists x, but is not true in all other models of the List Append axioms. The equation $\mathit{black}\ x = x$ is true for all strings of beans x, but does not follow equationally from AC and the Grecian Urn rules.

Definition 38 Cover Set. Let R be a rewrite system and $\succ$ be a reduction ordering containing its rewriting relation $\rightarrow$. A *cover set* S for conjecture c is a set of equations such that every ground instance $c\tau$ is equal (in the equational theory of R) to an instance $e\sigma$ of an equation e in S, such that $c\tau \succeq e\sigma$.

Definition 39 Rewriting Induction. Let R be a rewrite system and $\succ$ be a reduction ordering containing its rewriting relation $\rightarrow$. Let c be a conjecture and S, its cover set. *Rewriting induction* is the rule of inference which allows one to conclude that every ground instance of c is true for R by showing that each case e in S follows (equationally) from R and instances $c\sigma$ of the hypothesis that are smaller than e vis-à-vis $\succ$.

In practice, a conjecture $s=t$ is proved by expanding it into cases $s\sigma_1=t\sigma_1, \ldots, s\sigma_n=t\sigma_n$, for some "covering" substitutions $\sigma_1,\ldots,\sigma_n$, rewriting each case to be proved at least once using R, then attempting to construct rewrite proofs for them, freely using the inductive hypothesis $s=t$. The rewrite step guarantees that all instances of the hypothesis used in the rewrite proofs are smaller than the case of the conjecture under consideration. For ground convergent systems, narrowing at a particular position in a term is often a convenient method of generating cover sets and taking the requisite first step.

Theorem 40 [89]. *Rewriting induction is sound.*

Example 14 Reversal of Lists. Consider the following canonical system for reversing lists:

$$\begin{array}{rcl} reverse(nil) & \rightarrow & nil \\ reverse(cons(x,y)) & \rightarrow & append(reverse(y), cons(x, nil)) \\ append(nil, x) & \rightarrow & x \\ append(x, nil) & \rightarrow & x \\ append(cons(x,y), z) & \rightarrow & cons(x, append(y,z)) \\ append(append(x,y), z) & \rightarrow & append(x, append(y,z)) \ , \end{array}$$

and suppose one wishes to prove that

$$reverse(append(x,y)) \quad = \quad append(reverse(y), reverse(x)) \ .$$

All cases are covered by $x \mapsto nil$ and $x \mapsto cons(x_1, x_2)$, since ground terms containing *reverse* or *append* reduce to lists built of *cons* and *nil*. The corresponding two instances of the conjecture form a cover set. For $x = nil$, we need $reverse(y) = reverse(y)$, which is trivially the case. For $x = cons(x_1, x_2)$, we need to show

$$\begin{array}{c} append(reverse(append(x_2, y)), cons(x_1, nil)) \quad = \\ append(reverse(y), append(reverse(x_2), cons(x_1, nil))) \ . \end{array}$$

The left side rewrites by hypothesis to

$$append(append(reverse(y), reverse(x_2)), cons(x_1, nil)) \ ,$$

which rewrites to the right-hand side. □

In more complicated cases, additional lemmata may need to be added to R along the way. Completion can help find them.

Unlike traditional inductive proof methods, rewriting induction is guaranteed to disprove a false conjecture. (Of course, no method can prove all true ones.)

Definition 41 Ground Reducibility. A term s is *ground reducible* with respect to a rewrite system if all its ground instances $s\gamma$ are rewritable.

Theorem 42 [80][84]. *Ground reducibility is decidable for finite rewrite systems, but undecidable for AC systems.*

Theorem 43 [34][88][37]. *An equation c is not an inductive theorem of a ground convergent rewrite system R if and only if deducing critical pairs by unifying left-hand sides of rules in R on the larger (rather, not necessarily smaller) side of conjectures derived in this way from c produces an equation $u=v$ such that the larger of u and v is ground irreducible by R, or else u and v are distinct, but neither is ground reducible.*

The set of critical pairs between R and c is always a (generous) cover set, unless c is obviously not an inductive theorem, as described in the above theorem. Simplification by rules and lemmas may be incorporated into the proof procedure.

11 Future Research

Rewriting is an active field of theoretical and applied research. Current research topics include the following:

Typed Rewriting Under reasonable assumptions, virtually everything we have done extends to the multisorted case. Adding subsorts allows functions to be completely defined without having to introduce error elements for when they are applied outside their intended domains. But deduction in such "order-sorted" algebras presents some difficulties. The most popular approach is to insist that the sort of the right-hand side is always contained in that of the left.

AC termination The methods we have described for proving termination of ordinary rewriting are of only limited applicability when associativity and commutativity are built into the rewriting process. Special techniques have been devised to handle this case, which is of great practical importance.

Higher-order rewriting In the previous sections, we worked with first-order terms only. Since the typed lambda calculus is terminating and confluent, some researchers have been looking at ways of combining it with first-order rewriting in such a way as to preserve convergence, thereby endowing rewriting with higher-order capabilities.

Hierarchical systems From the point of view of software engineering, it is important that properties of rewrite programs, like termination and confluence, be modular. That is, we would like to be able to combine two terminating systems, or two convergent systems, and to have the same properties hold for the combined system. This is not true in general, not even when one system makes no reference to the function symbols and constants used in the other. Finding useful cases when systems may safely be combined is a current area of study.

Concurrency Confluent systems, in general, and orthogonal systems, in particular, are natural candidates for parallel processing, since rewrites at different positions are more or less independent of each other. Work is being undertaken on language and implementation issues raised by this possibility.

Infinite rewriting Rewriting can be extended to apply to structures other than the finite terms we have considered. Indeed, graph rewriting has important applications, since graphs allow one to represent structure-sharing, as well as infinite terms.

First-order theorem proving In Sect. 10, we saw how to simulate resolution-like inference equationally. A productive area of research is the application of ideas from rewriting to more traditional refutational theorem provers. Using orderings on terms and formulæ helps restrict deduction and increase the amount of simplification and redundancy elimination that can be incorporated without forfeiting completeness.

Acknowledgments

I thank Jieh Hsiang for his comments. This work was supported in part by the National Science Foundation under Grants CCR-90-07195 and CCR-90-24271 and by a Lady Davis fellowship at the Hebrew University of Jerusalem.

Selected Bibliography

The hundred items in this bibliography were chosen from among English language references on rewriting. They are divided into sections, corresponding to Sections 1–11 above, and are listed chronologically within section. (Beware that chronology oftentimes does not reflect the dependency of ideas, particularly as only the most polished versions are listed here.) Included are most books and surveys, plus many articles that were historically important, represent major stepping stones, or present the current state of research. Surveys are starred. Related topics, including the Lambda Calculus, Combinatory Logic, unification theory, and resolution theorem proving, have been omitted, as have descriptions of implementations. Several new books and surveys are in the offing.

1. Rewriting

1. Saul Gorn. Handling the growth by definition of mechanical languages. In *Spring Joint Computer Conference*, pages 213–224, Philadelphia, PA, Spring 1967.

*2. Gérard Huet and Derek C. Oppen. Equations and rewrite rules: A survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, New York, 1980.

*3. David R. Musser and Deepak Kapur. Rewrite rule theory and abstract data type analysis. In Jacques Calmet, editor, *Proceedings of the European Computer Algebra Conference (Marseille, France)*, volume 144 of *Lecture Notes in Computer Science*, pages 77–90, Berlin, April 1982. Springer-Verlag.

*4. Jean-Pierre Jouannaud and Pierre Lescanne. Rewriting systems. *Technology and Science of Informatics*, 6(3):181–199, 1987. French version: "La réécriture", *Technique et Science de l'Informatique* (1986), vol. 5, no. 6, pp. 433–452.

5. Benjamin Benninghofen, Susanne Kemmerich, and Michael M. Richter. *Systems of Reductions*, volume 277 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1987.

6. Matthias Jantzen. *Confluent String Rewriting*, volume 14 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.

*7. Jürgen Avenhaus and Klaus Madlener. Term rewriting and equational reasoning. In R. B. Banerji, editor, *Formal Techniques in Artificial Intelligence: A Sourcebook*, pages 1–41. Elsevier, Amsterdam, 1990.

*8. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Methods and Semantics, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.

*9. W. Wechler. *Reductions*, volume 25 of *EATCS Monographs on Theoretical Computer Science*, chapter 2, pages 89–133. Springer-Verlag, Berlin, 1991.

*10. 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 2, chapter 1, pages 1–117. Oxford University Press, Oxford, 1992.

2. Termination

11. Zohar Manna and Steven Ness. On the termination of Markov algorithms. In *Proceedings of the Third Hawaii International Conference on System Science*, pages 789–792, Honolulu, HI, January 1970.

12. Gérard Huet and Dallas S. Lankford. On the uniform halting problem for term rewriting systems. Rapport laboria 283, Institut de Recherche en Informatique et en Automatique, Le Chesnay, France, March 1978.

13. Dallas S. Lankford. On proving term rewriting systems are Noetherian. Memo MTP-3, Mathematics Department, Louisiana Tech. University, Ruston, LA, May 1979. Revised October 1979.

14. Sam Kamin and Jean-Jacques Lévy. Two generalizations of the recursive path ordering. Unpublished note, Department of Computer Science, University of Illinois, Urbana, IL, February 1980.

15. Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.

*16. Nachum Dershowitz. Termination of rewriting. *J. Symbolic Computation*, 3(1&2):69–115, February/April 1987. Corrigendum: *4*, 3 (December 1987), 409–410.

17. Hubert Comon. Solving inequations in term algebras (Preliminary version). In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 62–69, Philadelphia, PA, June 1990.

18. Jean-Pierre Jouannaud and Mitsuhiro Okada. Satisfiability of systems of ordinal notations with the subterm property is decidable. In J. Leach Albert, B. Monien, and M. Rodríguez Artalejo, editors, *Proceedings of the Eighteenth EATCS Colloquium on Automata, Languages and Programming (Madrid, Spain)*, volume 510 of *Lecture Notes in Computer Science*, pages 455–468, Berlin, July 1991. Springer-Verlag.
19. Nachum Dershowitz and Charles Hoot. Topics in termination. In C. Kirchner, editor, *Proceedings of the Fifth International Conference on Rewriting Techniques and Applications (Montreal, Canada)*, Lecture Notes in Computer Science, Berlin, June 1993. Springer-Verlag.

3. Confluence

20. M. H. A. Newman. On theories with a combinatorial definition of 'equivalence'. *Annals of Mathematics*, 43(2):223–243, 1942.
21. A. Selman. Completeness of calculii for axiomatically defined classes of algebras. *Algebra Universalis*, 2:20–32, 1972.
22. Barry Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the Association for Computing Machinery*, 20(1):160–187, January 1973.
23. J. Staples. Church-Rosser theorem for replacement systems. In J. Crossley, editor, *Algebra and Logic*, volume 450 of *Lecture Notes in Mathematics*, pages 291–307, Berlin, West Germany, 1975. Springer-Verlag.
*24. George M. Bergman. The Diamond Lemma for ring theory. *Advances in Mathematics*, 29(2):178–218, August 1978.
25. Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. of the Association for Computing Machinery*, 27(4):797–821, October 1980.
26. Jan Willem Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 1980.
27. Yoshihito Toyama. On the Church-Rosser property for the direct sum of term rewriting systems. *J. of the Association for Computing Machinery*, 34(1):128–143, January 1987.

4. Completion

28. Donald E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, U. K., 1970. Reprinted in *Automation of Reasoning 2*, Springer-Verlag, Berlin, pp. 342–376 (1983).
29. Yves Métivier. About the rewriting systems produced by the Knuth-Bendix completion algorithm. *Information Processing Letters*, 16(1):31–34, January 1983.
30. Phillipe Le Chenadec. *Canonical Forms in Finitely Presented Algebras*. Pitman-Wiley, London, 1985.
*31. Bruno Buchberger. History and basic features of the critical-pair/completion procedure. *J. Symbolic Computation*, 3(1&2):3–38, February/April 1987.
32. Deepak Kapur, D. R. Musser, and P. Narendran. Only prime superpositions need be considered for the Knuth-Bendix procedure. *J. Symbolic Computation*, 4:19–36, August 1988.
33. Leo Bachmair, Nachum Dershowitz, and David A. Plaisted. Completion without failure. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, chapter 1, pages 1–30. Academic Press, New York, 1989.
*34. Nachum Dershowitz. Completion and its applications. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, chapter 2, pages 31–86. Academic Press, New York, 1989.
35. Pierre Lescanne. Completion procedures as transition rules + control. In M. Diaz and F. Orejas, editors, *Proceedings of the Conference on Theory and Practice of Software Development*, volume 351 of *Lecture Notes in Computer Science*, pages 28–41, Berlin, 1989. Springer-Verlag.
*36. A. J. J. Dick. An introduction to Knuth-Bendix completion. *Computing Journal*, 34(1):2–15, February 1991.

37. Leo Bachmair. *Canonical Equational Proofs*. Birkhäuser, Boston, 1991.
*38. David A. Duffy. *Knuth-Bendix Completion*, chapter 7, pages 147–175. Wiley, Chichester, 1991.
39. Ursula Martin and Michael Lai. Some experiments with a completion theorem prover. *J. Symbolic Computation*, 13(1):81–100, January 1992.
40. Leo Bachmair and Nachum Dershowitz. Equational inference, canonical proofs, and proof orderings. *J. of the Association for Computing Machinery*, to appear.

5. Validity

41. T. Evans. On multiplicative systems defined by generators and relations, I. *Proceedings of the Cambridge Philosophical Society*, 47:637–649, 1951.
42. J. R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *J. of the Association for Computing Machinery*, 21(4):622–642, 1974.
43. Gérard Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *J. Computer and System Sciences*, 23(1):11–21, 1981.
44. Jieh Hsiang and Michaël Rusinowitch. On word problems in equational theories. In T. Ottmann, editor, *Proceedings of the Fourteenth EATCS International Conference on Automata, Languages and Programming (Karlsruhe, West Germany)*, volume 267 of *Lecture Notes in Computer Science*, pages 54–71, Berlin, July 1987. Springer-Verlag.
45. Gerald E. Peterson. Solving term inequalities. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 258–263, Boston, MA, July 1990.
*46. Nachum Dershowitz. Rewriting methods for word problems. In M. Ito, editor, *Words, Languages & Combinatorics (Proceedings of the International Colloquium, Kyoto, Japan, August 1990)*, pages 104–118, Singapore, 1992. World Scientific.

6. Satisfiability

47. M. Fay. First-order unification in an equational theory. In *Proceedings of the Fourth Workshop on Automated Deduction*, pages 161–167, Austin, TX, February 1979.
48. Jean-Marie Hullot. Canonical forms and unification. In R. Kowalski, editor, *Proceedings of the Fifth International Conference on Automated Deduction (Les Arcs, France)*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334, Berlin, July 1980. Springer-Verlag.
49. Nachum Dershowitz and G. Sivakumar. Solving goals in equational languages. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems (Orsay, France)*, volume 308 of *Lecture Notes in Computer Science*, pages 45–55, Berlin, July 1987. Springer-Verlag.
50. W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. *J. Symbolic Computation*, 7(3 & 4):295–318, 1989.
51. Alexander Bockmayr, Stefan Krischer, and Andreas Werner. An optimal narrowing strategy for general canonical systems. In M. Rusinowitch, editor, *Proceedings of the Third International Workshop on Conditional Rewriting Systems (Pont-a-Mousson, France, July 1992)*, volume 656 of *Lecture Notes in Computer Science*, Berlin, January 1993. Springer-Verlag.

7. AC-Completion

52. Gerald E. Peterson and Mark E. Stickel. Complete sets of reductions for some equational theories. *J. of the Association for Computing Machinery*, 28(2):233–264, April 1981.
53. Jean-Pierre Jouannaud and Hélène Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. on Computing*, 15:1155–1194, November 1986.
54. Leo Bachmair and Nachum Dershowitz. Completion for rewriting modulo a congruence. *Theoretical Computer Science*, 67(2 & 3), October 1989.

55. Hantao Zhang and Deepak Kapur. Unnecessary inferences in associative-commutative completion procedures. *Mathematical Systems Theory*, 23:175–206, 1990.

8. Conditional Rewriting

56. J. A. Bergstra and Jan Willem Klop. Conditional rewrite rules: Confluency and termination. *J. of Computer and System Sciences*, 32:323–362, 1986.
57. Stéphane Kaplan. Simplifying conditional term rewriting systems: Unification, termination and confluence. *J. Symbolic Computation*, 4(3):295–334, December 1987.
58. Peter Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.
59. Nachum Dershowitz, Mitsuhiro Okada, and G. Sivakumar. Canonical conditional rewrite systems. In *Proceedings of the Ninth Conference on Automated Deduction (Argonne, IL)*, volume 310 of *Lecture Notes in Computer Science*, pages 538–549, Berlin, May 1988. Springer-Verlag.
60. Stéphane Kaplan and Jean-Luc Rémy. Completion algorithms for conditional rewriting systems. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, pages 141–170. Academic Press, Boston, 1989.
*61. Jan Willem Klop and Roel C. de Vrijer. Extended term rewriting systems. In S. Kaplan and M. Okada, editors, *Proceedings of the Second International Workshop on Conditional and Typed Rewriting Systems (Montreal, Canada, June 1990)*, volume 516 of *Lecture Notes in Computer Science*, pages 26–50, Berlin, 1991. Springer-Verlag.
62. Peter Padawitz. Reduction and narrowing for Horn clause theories. *Computing Journal*, 34(1):42–51, February 1991.
63. Harald Ganzinger. A completion procedure for conditional equations. *J. Symbolic Computation*, 11:51–81, 1991.

9. Programming

64. Michael J. O'Donnell. *Computing in systems described by equations*, volume 58 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1977.
65. John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, December 1978.
66. Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and José Meseguer. Principles of OBJ2. In Brian Reid, editor, *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66, New Orleans, LA, January 1985. Association for Computing Machinery.
67. Nachum Dershowitz. Computing with rewrite systems. *Information and Control*, 64(2/3):122–157, May/June 1985.
68. Michael J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, Cambridge, MA, 1985.
69. Uday S. Reddy. On the relationship between logic and functional languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 3–36. Prentice-Hall, Englewood Cliffs, NJ, 1986.
70. Joseph A. Goguen and José Meseguer. EQLOG: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations, and Equations*, pages 295–363. Prentice-Hall, Englewood Cliffs, NJ, 1986.
71. Nachum Dershowitz and David A. Plaisted. Equational programming. In J. E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11: The logic and acquisition of knowledge*, chapter 2, pages 21–56. Oxford Press, Oxford, 1988. To be reprinted in *Logical Foundations of Machine Intelligence*, Horwood.
72. Steffen Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, West Germany, 1989.

73. Nachum Dershowitz and Mitsuhiro Okada. A rationale for conditional equational programming. *Theoretical Computer Science*, 75:111–138, 1990.
74. Gérard Huet and Jean-Jacques Lévy. Computations in orthogonal rewriting systems, I and II. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 395–443. MIT Press, Cambridge, MA, 1991.

10. Theorem Proving

75. Dallas S. Lankford. Canonical inference. Memo ATP-32, Automatic Theorem Proving Project, University of Texas, Austin, TX, December 1975.
76. Dallas S. Lankford and A. Michael Ballantyne. The refutation completeness of blocked permutative narrowing and resolution. In *Proceedings of the Fourth Workshop on Automated Deduction*, pages 53–59, Austin, TX, February 1979.
77. David R. Musser. On proving inductive properties of abstract data types. In *Proceedings of the Seventh ACM Symposium on Principles of Programming Languages*, pages 154–162, Las Vegas, NV, 1980.
78. Gérard Huet and Jean-Marie Hullot. Proofs by induction in equational theories with constructors. *J. of Computer and System Sciences*, 25:239–266, 1982.
79. Gerald E. Peterson. A technique for establishing completeness results in theorem proving with equality. *SIAM J. on Computing*, 12(1):82–100, February 1983.
80. David A. Plaisted. Semantic confluence tests and completion methods. *Information and Control*, 65(2/3):182–215, May/June 1985.
81. Jieh Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, 25:255–300, March 1985.
82. Etienne Paul. Equational methods in first order predicate calculus. *J. Symbolic Computation*, 1(1):7–29, March 1985.
83. Deepak Kapur and David R. Musser. Proof by consistency. *Artificial Intelligence*, 31(2):125–157, February 1987.
84. Deepak Kapur, Paliath Narendran, and Hantao Zhang. On sufficient completeness and related properties of term rewriting systems. *Acta Informatica*, 24(4):395–415, August 1987.
85. Jean-Pierre Jouannaud and Emmanuel Kounalis. Automatic proofs by induction in equational theories without constructors. *Information and Computation*, 81(1):1–33, 1989.
86. Wolfgang Küchlin. Inductive completion by ground proof transformation. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, pages 211–244. Academic Press, New York, 1989.
87. Hubert Comon and Pierre Lescanne. Equational problems and disunification. *J. Symbolic Computation*, 7:371–425, 1989.
88. Laurent Fribourg. A strong restriction of the inductive completion procedure. *J. Symbolic Computation*, 8(3):253–276, 1989.
89. Uday S. Reddy. Term rewriting induction. In M. Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction (Kaiserslautern, West Germany)*, volume 449 of *Lecture Notes in Computer Science*, Berlin, July 1990. Springer-Verlag.
90. Nachum Dershowitz and Eli Pinchover. Inductive synthesis of equational programs. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 234–239, Boston, MA, July 1990. AAAI.
91. Nachum Dershowitz. Ordering-based strategies for Horn clauses. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 118–124, Sydney, Australia, August 1991.
92. Deepak Kapur, Paliath Narendran, and Hantào Zhang. Automating inductionless induction using test sets. *J. Symbolic Computation*, 11:83–112, 1991.

11. Future Research

93. J. A. Goguen, C. Kirchner, and J. Meseguer. Concurrent term rewriting as a model of computation. In R. Keller and J. Fasel, editors, *Proceedings of Graph Reduction Workshop (Santa Fe, NM)*, volume 279 of *Lecture Notes in Computer Science*, pages 53–93. Springer-Verlag, 1987.
94. Yoshihito Toyama, Jan Willem Klop, and Hendrik Pieter Barendregt. Termination for the direct sum of left-linear term rewriting systems. In Nachum Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications (Chapel Hill, NC)*, volume 355 of *Lecture Notes in Computer Science*, pages 477–491, Berlin, April 1989. Springer-Verlag.
*95. A. J. J. Dick and P. Watson. Order-sorted term rewriting. *Computing Journal*, 34(1):16–19, February 1991.
96. Nachum Dershowitz, Stéphane Kaplan, and David A. Plaisted. Rewrite, rewrite, rewrite, rewrite, rewrite,.... *Theoretical Computer Science*, 83(1):71–96, 1991.
*97. Nachum Dershowitz, Jean-Pierre Jouannaud, and Jan Willem Klop. Open problems in rewriting. In R. Book, editor, *Proceedings of the Fourth International Conference on Rewriting Techniques and Applications (Como, Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 445–456, Berlin, April 1991. Springer-Verlag.
98. Jean-Pierre Jouannaud and Mitsuhiro Okada. Executable higher-order algebraic specification languages. In *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science*, 1991.
99. Leo Bachmair. Associative-commutative reduction orderings. *Information Processing Letters*, 43(1):21–27, August 1992.
*100. Jieh Hsiang, Helene Kirchner, Pierre Lescanne, and Michael Rusinowitch. The term rewriting approach to automated theorem proving. *J. Logic Programming*, 14(1&2):71–99, October 1992.