INFORMATION AND CONTROL 65, 122–157 (1985)

# Computing with Rewrite Systems

NACHUM DERSHOWITZ*

*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801*

Term-rewriting systems, that is, sets of directed equations, provide a paradigm of computation with particularly simple syntax and semantics. Rewrite systems may be used for straightforward computation by simplifying terms. We show how, in addition, restricted forms of the Knuth–Bendix "completion" procedure may be used to interpret logic programs written as a set of equivalence-preserving rewrite rules. We discuss verification issues and also illustrate the use of the full completion procedure to synthesize rewrite programs from specifications. © 1985 Academic Press, Inc.

## 1. INTRODUCTION

Term-rewriting systems (also known as "production systems" or "rewrite-rule systems") have been widely used for computation in formula-manipulation and theorem-proving systems. They serve as a general-purpose nondeterministic programming language possessing convenient mathematical properties (see, e.g., Burstall, MacQueen, and Sannella, 1980; Futatsugi, Goguen, Jouannaud, and Meseguer, 1984). Such programs are easy to understand, as they have a very simple syntax and semantics, based on equalities, with no explicit control.

In this paper we show how such systems may be used to compute in more general settings. The *completion procedure* (Knuth and Bendix, 1970), introduced as a means of deriving canonical term-rewriting systems to serve as decision procedures for given equational theories, may be used to interpret *logic programs* (Kowalski, 1974) written as a set of equivalence-preserving rewrite rules. Prolog (Clocksin and Mellish, 1981) is one successful attempt at combining the generality of predicate calculus with heuristic approaches to problem solving in an efficient programming

* This research was supported in part by the National Science Foundation under Grant MCS-83-07755 and by The Aerospace Corporation. It was in part pursued while the author was at Bar-Ilan University, on leave from the University of Illinois, and in part while at The Aerospace Corporation. This paper is a revision of "Equations as programming language," presented at the Fourth Jerusalem Conference on Information Technology (May 1984).

0019-9958/85 $3.00

language. Unlike Prolog, our method is not restricted to Horn clauses and allows one to incorporate equality between terms in a natural way. We use the completion procedure to generate new rewrite rules from existing rules that overlap. For the purposes of computation, the procedure may be restricted to an analogue of "linear-input" resolution (Kowalski, 1979), which is used in Prolog. Similar use of "narrowing" (Slagle, 1974) for computation is made in the language Eqlog (Goguen and Meseguer, 1984), which also provides for types and subsorts.

A rewrite system is a nondeterministic pattern-directed program that takes a term as input and returns a term as output. For example, the system

| *Monoid* | | |
|---|---|---|
| $x+0$ | $\to$ | $x$ |
| $0+x$ | $\to$ | $x$ |
| $x+(y+z)$ | $\to$ | $(x+y)+z$ |

where $x$, $y$, and $z$ are variables, eliminates zero summands from a sum, and reparenthesizes by associating to the left. Rules may be applied to any matching subterm; if the term $t'$ is the result of applying a rule $l \to r$ to a term $t$, we write $t \Rightarrow t'$. Applying the rules to the term $t=(a+b)+(0+(d+e))$, we get (among other possibilities)

$$t \Rightarrow (a+b)+(d+e) \Rightarrow ((a+b)+d)+e$$

or

$$t \Rightarrow (a+b)+((0+d)+e) \Rightarrow ((a+b)+(0+d))+e$$
$$\Rightarrow (((a+b)+0)+d)+e \Rightarrow ((a+b)+d)+e.$$

In both cases no further applications of the rules are possible.

The completion procedure (Knuth and Bendix, 1970) was originally suggested as a means of generating rewrite systems that can be used to decide the validity of identities in equational theories. A system $R$ is used as a decision procedure for a theory $E$ when it has the property that an equation $M=N$ is valid in $E$, if and only if applying rules in $R$ to $M$ and $N$, until no rule is applicable, results in the same (identical) term. Such a rewrite system $R$ is called a *canonical* rewrite system for the theory $E$. Given a finite set $E$ of equations (axioms) defining a theory and a (monotonic) well-founded (strict partial) ordering $\succ$ on the terms of the theory, the procedure attempts to find a finite canonical system $R$. Each rule $l \to r$ generated by the procedure is a *reduction* with respect to $\succ$, meaning that $t \succ t'$ whenever $t \Rightarrow t'$. And each rule is *sound* for $E$, i.e., the equation $l = r$ is valid in $E$.

For example, the above canonical system decides the validity of equations in the theory of monoids (semigroups with identity). Given the axioms

$$x + 0 = 0 + x$$

$$x = 0 + x$$

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

for example, and an appropiate ordering, the completion procedure will generate the above systems. An appropriate ordering, in this case, might compare lengths, and sum the relative positions of left parentheses for equal-length terms. The rules are all reductions, since each rule application either reduces the length or else the sum; the rules are all sound since they maintain equality in the theory. The system may be used to determine if two terms are equal by repeatedly applying the rules to both terms as long as possible. If the two terms rewrite to the identical term, then, and only then, are they equal in the theory.

In this paper we show how the Knuth–Bendix completion procedure may be used to execute logic-programs written as a set of equivalence-preserving rules. For example, the following is a rewrite program for appending two lists:

*List Append*

| | | |
|---|---|---|
| $append(x \cdot X, Y, x \cdot Z)$ | $\to$ | $append(X, Y, Z)$ |
| $append(nil, Y, Y)$ | $\to$ | $true$ |
| $append(Y, nil, Y)$ | $\to$ | $true$ |

where *nil* is the empty list and $\cdot$ adds a single element to the head of a list. The completion procedure, given this program and the goal rule

$$append(a \cdot b \cdot c \cdot nil,\ d \cdot e \cdot nil,\ W) \quad \to \quad answer(W),^1$$

[1] Parentheses have been omitted; $a \cdot b \cdot c \cdot nil$ is short for $a \cdot (b \cdot (c \cdot nil))$.

will generate the computation sequence

$$\begin{array}{rcl} append(b\cdot c\cdot nil,\ d\cdot e\cdot nil,\ W) & \to & answer(a\cdot W) \\ append(c\cdot nil,\ d\cdot e\cdot nil,\ W) & \to & answer(a\cdot b\cdot W) \\ append(nil,\ d\cdot e\cdot nil,\ W) & \to & answer(a\cdot b\cdot c\cdot W) \\ answer(a\cdot b\cdot c\cdot d\cdot e\cdot nil) & \to & true, \end{array}$$

giving the answer $a\cdot b\cdot c\cdot d\cdot e\cdot$nil to the question, "Which $W$ satisfies the goal relation $append(a\cdot b\cdot c\cdot nil,\ d\cdot e\cdot nil,\ W)$?" As we shall see, the two uses of rewrite systems, for Lisp-like computation by simplification and for Prolog-like computation by completion, may be combined in a single program.

Related applications of the Knuth–Bendix completion procedure to theorem proving include (Musser, 1980; Goguen, 1980; Huet and Hullot, 1981; Lankford, 1981; Hsiang, 1982; Kirchner, 1984; Hsiang and Dershowitz, 1983); related work on satisfiability procedures includes (Lankford, 1975; Fay, 1979; Lankford and Ballantyne, 1979; Hullot, 1980; Jouannaud, Kirchner and Kirchner, 1983). A unifying treatment of such applications may be found in (Dershowitz, 1982b) and a survey of equations and rewrite rules in (Huet and Oppen, 1980). Three implementations of the procedure are FORMEL (Hullot, 1980b), REVE (Lescanne, 1983), and RRL (Kapur and Sivakumar, 1983).

We also show how the full completion procedure may be used to reason about programs within the general first-order predicate calculus, using specifications and domain knowledge, themselves expressed as rewrite rules. In this way, the completion procedure can "compile" a complete program from a partial definition, verify the correctness of a program with respect to its specification (Dershowitz, 1982b), or automatically synthesize a rewrite-program from specifications, themselves expressed as rewrite rules (Dershowitz, 1985b). In synthesizing a program, the procedure itself does the "folding" (i.e., the introduction of recursive calls) based upon the axiomatization of the problem domain. If the completion procedure is given the right ordering, then it will find a program, if a program exists that does not require auxiliary definitions. When auxiliary functions are needed, their definition must be supplied by the programmer.

In the next section we define what it means for a rewrite system to work properly and survey verification methods. The main section, Section 3, shows how to use the completion procedure for computing in a rewrite-rule programming language. Section 4 illustrates how the procedure is used for program synthesis. We conclude with a brief discussion.

## 2. Rewrite Systems

A *term-rewriting* (*rewrite*) *system* $R$ over a set of terms $T$ is a finite set of rewrite rules, each of the form $l[\bar{x}] \rightarrow r[\bar{x}]$, where $l$ and $r$ are terms in $T$ containing variables $\bar{x}$. Such a rule may be applied to a term $t$ in $T$ if a subterm $s$ of $t$ matches the left-hand side with some substitution $\bar{\sigma}$ of terms for the variables appearing in $l$. The rule is applied by replacing that subterm $s = l[\bar{\sigma}]$ in $t$ with the corresponding right-hand side $r[\bar{\sigma}]$ of the rule, after making the same substitution of terms for variables in $r$.[2] The choice of which rule to apply where is made nondeterministically from amongst all possibilities. We write $t \Rightarrow t'$ to indicate that a term $t'$ (in $T$) is derivable from the term $t$ (in $T$) by a single application of some rule in $R$. By $t \overset{*}{\Rightarrow} t'$ we mean that $t'$ can be obtained from $t$ by any sequence of rule applications; if there is at least one application, we write $t \overset{+}{\Rightarrow} t'$. We say that $t'$ is a *normal form* of a term $t$, if $t \overset{*}{\Rightarrow} t'$ and no rule is applicable to $t'$.

For example, the following system (from Knuth, 1968) symbolically differentiates an expression with respect to $x$:

*Symbolic Differentiation*

| | | |
|---|---|---|
| $D_x x$ | $\rightarrow$ | $1$ |
| $D_x a$ | $\rightarrow$ | $0$ |
| $D_x(\alpha + \beta)$ | $\rightarrow$ | $D_x\alpha + D_x\beta$ |
| $D_x(\alpha - B)$ | $\rightarrow$ | $D_x\alpha - D_x\beta$ |
| $D_x(-\alpha)$ | $\rightarrow$ | $-D_x\alpha$ |
| $D_x(\alpha\beta)$ | $\rightarrow$ | $\beta\, D_x\alpha + \alpha\, D_x\beta$ |
| $D_x\left(\frac{\alpha}{\beta}\right)$ | $\rightarrow$ | $\frac{D_x\alpha}{\beta} - \alpha\frac{D_x\beta}{\beta^2}$ |
| $D_x(\ln\alpha)$ | $\rightarrow$ | $\frac{D_x\alpha}{\alpha}$ |
| $D_x(\alpha^\beta)$ | $\rightarrow$ | $\beta\alpha^{\beta-1}\, D_x\alpha + \alpha^\beta(\ln\alpha)\, D_x\beta$ |

where $D_x$ is the differentiation operator and $a$ stands for any constant symbol other than $x$. (Here $\alpha$ and $\beta$ are variables of the rewrite system and match any term, while $x$ is a constant of the system and matches only itself.

[2] We use the notation $t = u[s_1, \ldots, s_n]$ to mean that the term $t$ contains occurrences of subterms $s_1, \ldots, s_n$ embedded in the context $u$ (i.e., $u$ is $t$ without the $s_i$). At the same time, we use the informal notation $t[\bar{\sigma}]$ to denote the term obtained from $t[\bar{x}]$ by applying the substitution $\bar{\sigma}$, replacing all occurrences of the variables $\bar{x}$ in $t$ with the corresponding elements of $\bar{\sigma}$.

In the second rule, $a$ matches constants like 0, 1, $y$, and so on.) In this manner, rewrite rules have long been used for ad-hoc computation in symbol manipulation systems, such as REDUCE (Hearn, 1971), for simplifying in theorem provers (e.g., Waldinger and Levitt, 1974), and in conjunction with abstract data types, as in AFFIRM (Gerhart *et al.*, 1980). Rewrite systems (or even semi-Thue systems) have the full computational power of Turing machines. (See Huet and Lankford, 1978, for one proof.)

Five desirable properties involved in the verification of rewrite systems are:

(1) *termination*—no infinite derivations are possible,

(2) *confluence*—each term has at most one normal form,

(3) *soundness*—terms are only rewritten to equal terms,

(4) *completeness*—equal terms have the same normal form,

(5) *correctness*—all normal forms satisfy given requirements.

(More precise definitions are given below.) A rewrite system $R$ is *canonical* for an equational theory $E$, if it is terminating, confluent, sound with respect to $E$, and complete with respect to $E$. If $R$ is a canonical system for a theory $E$, then it can be used to decide whether an equation $M = N$ follows from the axioms in $E$ by checking whether or not the unique normal forms of $M$ and $N$ are the same.

In this section we survey some methods for establishing the above properties. First we consider the "intrinsic" properties, termination and confluence. Then we consider soundness and completeness, which are properties relative to a notion of "equality," and finally we discuss correctness, which is relative to specified "requirements." Each of the five properties is in general undecidable. But, as we will see, confluence is decidable for terminating systems, and completeness is decidable for confluent, terminating systems. Correctness is shown to be decidable in certain common cases.

## 2.1. *Intrinsic Properties*

A system $R$ is said to *terminate* for a set of terms $T$ if there is no infinite derivation $t_1 \Rightarrow t_2 \Rightarrow t_3 \Rightarrow \cdots$ of terms $t_i$ in $T$. Termination is in general an undecidable property of rewrite systems (Huet and Lankford, 1978; Dershowitz, 1985).

The standard method of demonstrating termination is to use monotonic well-founded orderings on terms. A (strict) partial ordering $\succ$ over a set of terms $T$ is *monotonic* if $f(\cdots t \cdots) \succ f(\cdots t' \cdots)$ whenever $t \succ t'$, for all terms in $T$, that is, reducing any subterm reduces the whole term. It is *well founded* if it admits no infinite descending sequences of elements $t_1 \succ t_2 \succ t_3 \succ \cdots$.

THEOREM 1 (Manna and Ness, 1970). *A rewrite system $R$ over a set of terms $T$ terminates, if and only if there exists a monotonic well-founded ordering $\succ$ over $T$ such that $l \succ r$ for each rule $l \to r$ in $R$ and for any substitution of terms for its variables.*

Polynomial interpretations, mapping terms into the nonnegative integers, are commonly used for this purpose (Lankford, 1975).

Frequently, termination can be proved using the notion of "simplification ordering." A monotonic ordering $\succ$ over $T$ is a *simplification ordering* if it satisfies the additional condition $f(\cdots t \cdots) \succ t$ for all terms in $T$, that is, a term is greater than any of its subterms. If a symbol $f$ is allowed to take a variable number of arguments, then a simplification ordering $\succ$ must also satisfy $f(\cdots t \cdots) \succ f(\cdots \quad \cdots)$.

THEOREM 2 (Dershowitz, 1982). *A rewrite system $R$ over a set of terms $T$ terminates, if there exists a simplification ordering $\succ$ over $T$ such that $l \succ r$ for each rule $l \to r$ in $R$ and for any substitution of terms for its variables.*

To see that the differentiation example terminates, one can use the *recursive path ordering* (Dershowitz, 1982), with which terms are compared based upon an ordering on their function symbols. In this example, the differentiation operator $D_x$ is taken to be greater than any other, since it is the symbol being eliminated. Informally, when two terms $s$ and $t$ have the same outermost symbol, their respective subterms are compared (recursively); if the outermost symbol of $s$ is greater than that of $t$, then $s$ is compared with each of the subterms of $t$. Since the arguments to $D_x$ on the right-hand side of each rule are subterms of the argument on the left-hand side, and are therefore smaller in any simplification ordering, the left-hand side is always greater than the right-hand side in this ordering. For example,

$$D_x(\alpha\beta) > \alpha D_x\beta + \beta D_x\alpha,$$

since the left-hand side argument $\alpha\beta$ is greater than the right-hand side arguments $\beta$ and $\alpha$. Thus, the system terminates for terms containing an arbitrary number of differentiation operators. A survey of methods used for proving termination of rewrite systems may be found in (Dershowitz, 1985).

A rewrite system $R$ is said to be *confluent*, if whenever a term $s$ can be rewritten to two distinct terms $p$ and $q$, both $p$ and $q$ can be rewritten to some term $t$. Confluence is equivalent to the *Church–Rosser* property:

THEOREM 3 (Curry and Feys, 1958). *A rewrite system $R$ is confluent, if* *and only if whenever two terms $s$ and $t$ can be obtained one from the other by applying rules in $R$ in either direction (substituting an instance of the right-hand side for the corresponding left-hand side or vice versa), it is also the case that both $s$ and $t$ can be reduced by $R$ to the same term.*

If a system terminates, then every term has at least one normal form; if it is confluent, then there can be at most one normal form; if it is both terminating and confluent, then the system defines a unique irreducible normal form $R(t)$ for each term $t$. For many applications, confluence is desirable or necessary. For nonterminating systems, confluence is in general undecidable (see, e.g., Huet and Oppen, 1980).

To demonstrate confluence of terminating systems, we need the notion of "critical pair." Let $l[\bar{x}] \rightarrow r[\bar{x}]$ and $l'[\bar{y}] \rightarrow r'[\bar{y}]$ be two (not necessarily different) rules in $R$ whose variables $\bar{x}$ and $\bar{y}$ have been renamed, if necessary, so that they are distinct. We say that the left-hand side $l'$ *overlaps* (or *superposes*) the left-hand side $l$, if $l[\bar{x}]$ contains a (non-variable) subterm $s$ embedded in some context $u$—to indicate this we write $l[\bar{x}] = u[s][\bar{x}]$—such that there is a (most general) substitution $\bar{\sigma}$ for the variables $\bar{x}$ and $\bar{y}$ for which $s[\bar{\sigma}] = l'[\bar{\sigma}]$. If $l'$ overlaps $l$, then the overlapped term $l[\bar{\sigma}]$ can be rewritten to either $r[\bar{\sigma}]$ or $u[r'][\bar{\sigma}]$. The equation between these two possibilities, $r[\bar{\sigma}] = u[r'][\bar{\sigma}]$, is called a *critical pair.*

THEOREM 4 (Knuth and Bendix, 1970). *A terminating rewrite system is confluent, if and only if both terms in each of its critical pairs reduce to the same term.*

Since a rewrite system has only a finite number of critical pairs, confluence of terminating systems is decidable. The differentiation system, for example, has no critical pairs, since each left-hand side contains only one $D_x$ with a different operator as its argument. Thus, it is confluent, as well as terminating, and the result of applying the system to any term is unique. An alternative, "semantic" approach to confluence is pursued in (Plaisted, 1985).

## 2.2. *Relative Properties*

Given a set $E$ of equations (axioms), by the *equational theory* (*variety*) $E$, we mean the class of all models satisfying the equations in $E$ (and the axioms of equality). An equation $M = N$ is *valid* for $E$ if it is true in all models of $E$. An equation $M = N$ is *provable* from the axioms of $E$ if there exists a finite proof $P_1 = P_2 = \cdots = P_n$ $(n \geqslant 1)$ such that $P_1 = M$, $P_n = N$, and for each step $P_i = P_{i+1}$ $(i = 1,..., n-1)$, $P_{i+1}$ is obtained by replacing a subterm of $P_i$ that is an instance of one side of an axiom in $E$ by the corresponding instance of the axiom's other side ("replacing equals by equals"). By the following theorem, validity and provability coincide:

THEOREM 5 (Birkhoff, 1935). *An equation $M = N$ is valid for an equational theory E, if and only if it is provable from the axioms of E.*

A rewrite system $R$ is said to be *sound* for a given theory $E$ if each rule is a valid equation for the theory $E$, i.e., if each rule is true in all models of $E$. (The theory $E$ need not be equational for this definition to apply.) If we can show that each rule in $R$ follows from the axioms in $E$, then the system is demonstratably sound. But since not all (finite equational) theories are decidable (see Taylor, 1969), soundness of a system is in general undecidable. The differentiation example is sound with respect to differential calculus, since each rule is an equality of the calculus.

A rewrite system $R$ is said to be *complete* for a given axiomatization $E$ if any two terms that are equal in the theory can be rewritten by $R$ to the same term. Completeness is undecidable, since confluence is, and a system is confluent if and only if it is complete with respect to its own rules considered as equations (the Church–Rosser property). It is straightforward to see that

THEOREM 6. *A confluent rewrite system R is complete with respect to an equational theory E, if and only if both sides of each axiom in E rewrite under R to the same term.*

Thus, a terminating and confluent system $R$ is complete with respect to a theory $E$ if and only if both sides of each axiom in $E$ rewrite to the same normal form under $R$. A terminating and complete system $R$ provides a decision procedure for validity in an equational theory $E$. The differentiation example is *incomplete* with respect to the differential calculus, since it does not consider all functions, e.g., $D_x \sin x = \cos x$, though neither is reducible.

A rewrite system $R$ is said to be *correct* with respect to a set $\Phi$ of *input* terms and a set $\Psi$ of *output* terms if all the irreducible normal forms of terms in $\Phi$—denoted $R(\Phi)$—are in $\Psi$. If, in addition, all output terms are irreducible forms of input terms, then $\Psi = R(\Phi)$. This definition is inspired by (Plaisted, 1980). It corresponds to "partial correctness" of conventional programs (see Manna, 1974) and is similarly undecidable.

A *test set* $S$ for $R$ is a set of terms such that a necessary and sufficient condition for $\Psi$ to contain all of $R(\Phi)$ is for every term in $S$ to be reducible. Any term in a test set that contains an instance of another test term can be omitted (since the former is irreducible only if the latter is); in particular, all proper subterms of test terms should themselves be irreducible.

THEOREM 7. *Let $R$ be a rewrite system and suppose that $\Phi$ is closed under $R$, i.e., applying a rule to a term in $\Phi$ gives a term in $\Phi$. If $S$ is a set of nonoutput terms (i.e., a subset of $\Phi - \Psi$) such that all nonoutput terms in $\Phi$ contain an instance of a term in $S$, then $S$ is a test set for the correctness of $R$ with respect to $\Phi$ and $\Psi$.*

When a finite test set is available for a closed system, correctness can be decided. This theorem extends (Plaisted, 1980), where it is also pointed out that the whole of $\Phi - \Psi$ constitutes a test set.

*Proof.* Assume $S \subseteq \Phi - \Psi$ and all $t \in \Phi - \Psi$ are of the form $t[s[\bar{\sigma}]]$ for some $s \in S$. We need to show that $R(\Phi) \subseteq \Psi$ if and only if $S \cap R(\Phi) = \emptyset$. If $R(\Phi) \subseteq \Psi$, then $S \cap R(\Phi) \subseteq S \cap \Psi$. But $S \cap \Psi = \emptyset$, since $S \subseteq \Phi - \Psi$; thus, $S \cap R(\Phi) = \emptyset$. Suppose now that $R(\Phi) \not\subseteq \Psi$, i.e., there exists an irreducible nonoutput term $t$ in $R(\Phi) - \Psi$. By assumption, there is some $s \in S$ that has an instance $s[\bar{\sigma}]$ that is a subterm of $t$. But since $t \in R(\Phi)$ is irreducible, $s$ must also be. Hence, $S \cap R(\Phi) \neq \emptyset$. ▌

For example, imagine that we wanted to represent integers as sums of ones, possibly prefixed by a unary minus, and we wish to show that some rewrite system for addition, when applied to integers represented in this way, always results in such an integer. That is, the input terms $\Phi$ are all *ground* (i.e., variable-free) terms constructed from the constants 0 and 1, the unary negation operator $-$, and the binary addition operator $+$; the output terms $\Psi$ are

$$\{0,\ 1,\ 1+1,\ (1+1)+1, \cdots,\ -1,\ -(1+1), \cdots\}.$$

What we need then, is for the system to eliminate all occurrences of 0 from other terms, all second summands other than 1, and all nonoutermost negations. That suggests looking at the set

$$\{x+0,\ 0+x,\ -0,\ x+(y+z),\ x+(-y),\ - -x,\ (-x)+y\}.$$

If these seven terms are reducible, then the system in question is correct. They are not, however, a test set, since a system might, for example, reduce all *variable-free* instances of $(-x)+y$, yet not reduce the term $(-x)+y$ itself. If this is the case, one can try replacing the irreducible terms with less general instances. For example: expanding $(-x)+y$ to cover all possibilities for $y$ gives $(-x)+0$, $(-x)+1$, $(-x)+(-y)$, and $(-x)+(y+z)$. Eliminating the three redundant terms, containing instances of others, leaves only $(-x)+1$. This term, in turn, can be replaced by

$(-1)+1$ and $-(y+z)+1$, and the latter, by just $-(y+1)+1$. The following system is therefore correct with respect to the given specification:

| *Unary Integer Addition* | | |
|---|---|---|
| $n+0$ | $\rightarrow$ | $n$ |
| $0+n$ | $\rightarrow$ | $n$ |
| $-0$ | $\rightarrow$ | $0$ |
| $l+(m+n)$ | $\rightarrow$ | $(l+m)+n$ |
| $m+(-n)$ | $\rightarrow$ | $-((-m)+n)$ |
| $--n$ | $\rightarrow$ | $n$ |
| $(-1)+1$ | $\rightarrow$ | $0$ |
| $-(n+1)+1$ | $\rightarrow$ | $-n$ |

Consider, now, the following common situation: $F$ is a finite set of function symbols, some of which are designated *defined*, $G$ is the set of *ground* terms composed of symbols in $F$, and $C$ is the (nonempty) set of ground *constructor* terms containing no defined symbols. Let $R$ be a rewrite system, and suppose that we wish to show that $R$ reduces terms in $G$ to terms in $C$.[3] The following theorem provides a method of deciding if in fact $R(G)=C$; it extends results of (Huet and Hullot, 1980).

THEOREM 8. *The constructor terms in $C$ are irreducible by a rewrite system $R$, if and only if each left-hand side of $R$ contains a defined symbol. Furthermore, such a system $R$ has a finite test set $S$ for correctness with respect to ground terms $G$ and ground constructor terms $C$.*

The following proof gives a construction of the test set $S$; a more efficient construction of test sets is given in (Thiel, 1984).

*Proof.* If there is a left-hand side with no defined symbols, then substituting ground constructor terms for its variables yields a reducible ground constructor term. Conversely, if there is a reducible constructor term, then the rule that reduces it cannot have a defined symbol on its left-hand side.

Let $D$ be the set of all terms containing only one defined symbol, and

[3] Ground terms are closed under any terminating (unsorted) system.

that one as the outermost symbol; by assumption each left-hand side of $R$ contains a term in $D$. Clearly, every term in $G-C$ must have a subterm in $D \cap G$; in other words, the ground terms in $D$ are a (possibly infinite) test set. If $D \cap G$ is finite, then it can serve as the test set $S$. Otherwise, there must be an infinite number of ground constructor terms. Let $m$ be the maximum depth of left-hand sides of $R$ that are in $D$. Let $S$ be the set of those terms in $D$ that are of depth no greater than $m$ and with variables only at depth $m$. This $S$ suffices as a test set, since all larger ground terms in $D$ can be reduced by the same rule as some term of restricted depth.

To see this, suppose that some term $t$ in the test set $D \cap G$ is irreducible. Let $s$ be $t$ with all subterms below depth $m$ replaced with distinct variables. Were $s$ reducible, then the same rule would apply to its instance $t$. On the other hand, for any irreducible nonground term $s \in S$, consider the term $t \in D \cap G$ obtained by substituting distinct ground constructor terms for each variable in $s$. (These distinct ground terms should differ in depth by at least $m$.) It too would be irreducible, since no rule is applicable at a subterm of $t$, and rules applied at the outermost defined symbol of $t$ are too shallow for the substitution to matter. ▮

In our differentiation example, $D_x$ is the defined symbol, and all the other symbols are constructors. Each left-hand side has one $D_x$ as its outermost operator. Furthermore, there is a rule for $D_x$ applied to every other operator. Thus, the left-hand sides, themselves, suffice as a test set, demonstrating that the system eliminates all differentiation operators from any term (built from the defined operator $D_x$, constants, and the addition, subtraction, negation, multiplication, division, natural logarithm, and exponentiation operators).

For the purposes of using rewrite systems for logic programming, as described in the next section, the following correctness property is desirable: all ground terms (in $G$) equal (under $R$) to the particular term *true* should reduce to *true*. This property holds, in particular, if $R$ is ground confluent and *true* is irreducible. By *ground confluent*, we mean that two equal ground terms always reduce to the same term. Plaisted (1984) develops methods, related to output correctness, for demonstrating ground confluence. (See also Padawitz, 1983.)

## 2.3. *Associativity and Commutativity*

Associativity and commutativity of functions cannot be handled by including axioms for these properties as rules (without losing the termination property). Instead, special unification algorithms are used to take associativity and commutativity into account. As an example, consider the following canonical rewrite system for the propositional calculus (Watts and Cohen, 1980; Hsiang, 1982):

*Propositional Calculus*

| | | |
|---|---|---|
| $u \wedge true$ | $\rightarrow$ | $u$ |
| $u \wedge false$ | $\rightarrow$ | $false$ |
| $u \wedge u$ | $\rightarrow$ | $u$ |
| $u \oplus false$ | $\rightarrow$ | $u$ |
| $u \oplus u$ | $\rightarrow$ | $false$ |
| $(u \oplus v) \wedge w$ | $\rightarrow$ | $(u \wedge w) \oplus (v \wedge w)$ |
| $\neg u$ | $\rightarrow$ | $u \oplus true$ |
| $u \vee v$ | $\rightarrow$ | $(u \wedge v) \oplus u \oplus v$ |
| $u \supset v$ | $\rightarrow$ | $(u \wedge v) \oplus u \oplus true$ |

where $\wedge$ is "and," $\oplus$ is "exclusive-or," $\neg$ is "not," $\vee$ is "inclusive-or," and $\supset$ is "implies." Both $\wedge$ and $\oplus$ are implicitly associative and commutative. That means, for example, that the rule $u \wedge u \rightarrow u$ applied to $(p \wedge q) \wedge p$ yields $p \wedge q$. Since these functions are associative, there is no significance to the parenthesization, and accordingly terms may be "flattened" by removing embeddings of the same associative symbols, e.g., $(p \wedge q) \wedge p$ is written $p \wedge q \wedge p$.

The termination of this system can be shown using a recursive path ordering in which the three function symbols $\neg$, $\vee$, and $\supset$ are greater than $\wedge$, which is greater than $\oplus$, and which, in turn, is greater than the constant symbols *true* and *false* (the two constants are considered to be equivalent in the symbol ordering). The last three rules reduce a term by eliminating an occurrence of one of the greatest functions; the first four rules replace a term with a subterm and the next one replaces $\oplus$ with the lesser *false*; the sixth rule (for distributing $\wedge$ over $\oplus$) replaces a large conjunction with two smaller ones, connected by the lesser symbol $\oplus$.

However, since some of the functions are associative and commutative, one must show that the rules also reduce when they interact with such functions. Thus, for every rule $l \rightarrow r$ with an associative-commutative symbol $f$ outermost on one of the sides $l$ or $r$ (or just a variable on the right-hand side), we also need $f(l, \alpha) \succ f(r, \alpha)$, for all terms $\alpha$. For the propositional calculus system, one must have, for example, that

$$(u \oplus v) \wedge w \wedge \alpha \succ (u \wedge w \oplus v \wedge w) \wedge \alpha$$

under the given recursive path ordering. For this to be the case, the ordering must be modified so that terms are compared by comparing their fully distributed forms. With this change, all rules are reductions, except for distributivity itself, which can be shown to terminate independently. (This is an example of an "associative path ordering," see Plaisted, 1983; Plaisted and Bachmair, 1985. For a general discussion of "$AC$-termination," see Dershowitz *et al.*, 1983). For the use of "commutation properties" in such arguments, see Jouannaud and Muñoz, 1984; Bachmair and Dershowitz, 1985).

The propositional calculus system is confluent, since all its critical pairs reduce to the same term. When, as in this example, some of the functions on the left-hand sides are associative and commutative, then an associative–commutative unification algorithm (Fages, 1984; Livesey and Siekmann, 1976; Stickel, 1981) is used to find a substitution such that one left-hand side overlaps another. The definition of "overlap" must also be extended to include cases in which two rules have overlapping subterms of the same associative–commutative symbol (Lankford and Ballantyne, 1977b; Peterson–Stickel, 1981). To do this, *extended* rules, of the form $f(l, \alpha) \rightarrow f(r, \alpha)$, must also be considered for each rule whose left-hand side $l$ has an associative–commutative outermost symbol $f$. All resulting *extended* critical pairs must reduce to the same term, up to permutation of arguments of the associative–commutative symbols. For example, for the critical pair $(u \wedge w) \oplus (u \wedge w) = \mathit{false} \wedge w$, obtained by overlapping the fifth and sixth rules, we have $(u \wedge w) \oplus (u \wedge w) \Rightarrow \mathit{false} \Leftarrow \mathit{false} \wedge w$. We must also consider overlapped terms such as $u \oplus u \oplus \mathit{false}$ to which both $u \oplus u \rightarrow \mathit{false}$ and $u \oplus \mathit{false} \rightarrow u$ may be applied; the critical pair is $\mathit{false} \oplus \mathit{false} = u \oplus u$, both of which reduce to *false*. (General criteria for confluence in the presence of equations are given in Jouannaud and Kirchner, 1984.)

That this system is sound with respect to the propositional calculus follows from the fact that each rule is a propositional equivalence and $\wedge$ and $\oplus$ are in fact associative and commutative. It is also easily seen to be complete. The system is correct with respect to the set of all propositional formulae and the set of "sums" of conjunctions. To see this, we need to show that any term that is not a sum of conjunctions is reducible. But then it must either contain another symbol, in which case one of the last three rules can reduce it, or else it must contain a conjunction of a sum, in which case it is reducible by the sixth rule (distributivity). The system is also correct with respect to the set of ground terms and the truth-value constructors *true* and *false*. To see that all nonconstant ground terms must contain an instance of some left-hand side, note that each term in the set

$$\{\neg x,\ x \vee y,\ x \supset y,\ x \wedge \mathit{false},\ x \wedge \mathit{true},\ x \oplus \mathit{false},\ \mathit{true} \oplus \mathit{true}\}$$

is reducible. (A general method for testing correctness in the presence of equations is given in Kounalis and Zhang, 1985.)

Since the above system is canonical for propositional calculus, it provides a means of deciding validity of propositional equivalences. Thus, the system may be used to check for either validity or unsatisfiability of propositional formulae by reducing terms to their unique normal form: a formula that reduces to *true* is a *tautology*, one that reduces to *false* is a *contradiction*, while anything that reduces to neither is *contingent*.

## 3. Rewrite Programs

Rewrite systems may be used as "logic programs" (Kowalski, 1974), in addition to their straightforward use for computation by rewriting. The result is a Prolog-like programming language, the main differences being that rewrite rules are equivalences, rather than implications in Horn-clause form, and that the Knuth–Bendix completion (Knuth and Bendix, 1970) procedure acts as the interpreter, rather than resolution. Furthermore, this programming paradigm allows for the advantageous combination of simplification (by rewriting) and deduction (by completion). The idea of using a (resolution) theorem-prover for (loop-free) programming was suggested by (Green, 1969; Waldinger, 1969); Kowalski (1979) suggests that Horn-clause programs be specified as equivalences. Other deduction-based computing methods include (Bowen, 1982; Hansson, Haridi, and Tärnlund, 1982; Malachi, Manna, and Waldinger, 1984).

The following set of rules is an example of a *rewrite program*, designed to compute the quotient and remainder of two natural numbers:

| *Integer Division* | | |
|---|---|---|
| $div(x+y+1, y+1, q+1, r)$ | $\rightarrow$ | $div(x, y+1, q, r)$ |
| $div(x, x+z+1, 0, x)$ | $\rightarrow$ | $true$ |

Here $+$ is associative and commutative (with identity 0), and positive integers are represented in unary (in the form $1+1+\cdots+1$). Associativity and commutativity are needed so that $x+y$, for example, can be matched with $1+1+1+1$, with $x=y=1+1$.[4] The first rule is the recursive case (denominator not greater than numerator); the second is the base case (denominator is greater). To compute the quotient and remainder of two numbers $a$ and $b$ with this system, the rule

$$div(a, b, q, r) \quad \rightarrow \quad answer(q, r)$$

[4] One could construct a division program that does not need associative–commutative unification, but that would necessitate subprograms for unary comparison and subtraction.

is added, meaning that $q$ and $r$ are the answer if (and only if) they are the quotient and remainder, respectively, of $a$ and $b$. As we shall see, the completion procedure will then generate a rule

$$answer(c, d) \quad \rightarrow \quad true,$$

containing the answer values $c$ and $d$ for $q$ and $r$, respectively.

Logic does not distinguish between input and output. Just as the above program determines the quotient and remainder from numerator and denominator, it also determines which numerator-remainder pairs correspond to any given quotient and denominator. Thus, while some questions may have a unique answer, others may have many or none, and a particular program may be more effective for answering one type of question than another. In the remainder of this section, we will see how various forms of completion may be used to generate answers; we will concentrate on finding a single answer, though there may be many.

## 3.1. *The Completion Procedure*

To compute with a rewrite *program* in contrast to rewrite *systems*, the completion procedure (Knuth and Bendix, 1970) is used. The procedure takes as input a finite set $R$ of rules, a finite set $E$ of equations, and a program to compute a monotonic well-founded ordering $\succ$. Initially, $R$ may contain any set of sound reductions, all of whose critical pairs are in $E$. The procedure, shown below, then generates new rules, each of which is a sound reduction:

*Completion Procedure*

Repeat as long as equations are left in $E$. If none remain, terminate successfully.

(1) Remove an equation $M=N$ (or $N=M$) from $E$ such that $M \succ N$. If none exists, terminate with failure (abort).

(2) Add the rule $M \rightarrow N$ to $R$.

(3) Use $M \rightarrow N$ (followed by any rules in $R$) to reduce the right-hand sides of existing rules to their normal forms.

(4) Add to $E$ all critical pairs formed from $R$ using $M \rightarrow N$.

(5) Remove all the old rules from $R$ whose left-hand side contains an instance of $M$.

(6) Use $R$ to reduce both sides of equations in $E$ to their normal forms. Remove any equation that reduces to identity.

The original purpose of this procedure was to extend nonconfluent systems to confluent ones. When the completion procedure terminates successfully, it returns as output a canonical system for the given theory (consisting of the axioms in the input sets $E$ and the rules in $R$ considered as equations). If the procedure aborts, it may backtrack to the last choice made in step (1).[5] Nevertheless, it may be that a particular choice of ordering $\succ$ precludes finding a canonical system. The procedure may also go on generating an infinite number of new rules, never finding a canonical system and never aborting. In that case, we say that the procedure *loops*. By a *fair* execution of the procedure, we mean that no equation remains in $E$ forever, i.e., every orientable equation placed in $E$ will eventually be reduced to an instance of the identity axiom, $x=x$, or else turned into a rule. All critical pairs need not be added to $E$ immediately after each new rule is generated, as long as every pair is eventually taken into account. (See Huet, 1981, for details.)

The following theorem determines the extent to which completion is guaranteed to find proofs:

THEOREM 9 (Huet, 1981). *An equation $M=N$ is valid in an equational theory E, if and only if the completion procedure—given the equations E and monotonic well-founded ordering $\succ$—eventually will have generated enough rules for M and N to reduce to the identical term. This, provided that the procedure executes fairly and does not abort.*

This result also applies when some of the function symbols are implicitly associative and commutative (Lankford, 1981), and, generally, for implicit theories having complete unification algorithms (Jouannaud and Kirchner, 1984). Accordingly, we allow rewrite systems to contain such symbols, and use the extensions of the Knuth–Bendix procedure to commutative functions (Lankford and Ballantyne, 1977), to associative–commutative functions (Lankford and Ballantyne, 1977b; Peterson and Stickel, 1981), to associative–commutative–idempotent functions (Fages, 1983), and to associative–commutative functions with an identity (Fages, 1983).

## 3.2 *Computing*

To compute by completion, a *goal rule* is added to a rewrite system. Goal rules are of the form

$$p[\bar{s}, \bar{z}] \quad \rightarrow \quad answer(\bar{z}),$$

where $p$ is a *calling term* containing input values (i.e., irreducible ground

[5] Backtracking can make the difference between success and failure, see (Dershowitz and Marcus, 1984).

terms) $\bar{s}$ and output variables $\bar{z}$.[6] A rewrite program $R$ is said to *compute* the *input/output relation* $p[\bar{x}, \bar{z}]$ (the $\bar{x}$ are input variables), if the completion procedure, given any such a goal rule, will generate an *answer rule* of the form $answer(\bar{t}) \rightarrow true$, such that $p[\bar{s}, \bar{t}]$ is true—*without aborting* (provided such a $\bar{t}$ exists). The ordering supplied to the procedure should make *true* less than *answer* terms and *answer* terms less than any other term.

The division system is such a program. For example, to compute the quotient and remainder of 7 and 3, the rule

$$div(7, 3, q, r) \quad \rightarrow \quad answer(q, r)$$

is added. (The numerals in the above rule are just abbreviations for their unary representation as sums of ones, e.g., 3 is short for $1+1+1$.) Completion generates

$$div(4, 3, q, r) \quad \rightarrow \quad answer(q+1, r)$$

by overlapping the goal rule with the first program rule; using that rule again gives

$$div(1, 3, q, r) \quad \rightarrow \quad answer(q+2, r);$$

finally overlapping this subgoal with the second rule yields the answer rule

$$answer(2, 1) \quad \rightarrow \quad true.$$

Note that the same program may be used to compute other arguments of *div*. For example, to compute the product of 3 and 2, one adds the goal

$$div(x, 3, 2, 0) \quad \rightarrow \quad answer(x).$$

Completing generates

$$\begin{aligned} div(x, 3, 1, 0) \quad &\rightarrow \quad answerx + 3) \\ div(x, 3, 0, 0) \quad &\rightarrow \quad answer(x+6) \\ answer(6) \quad &\rightarrow \quad true. \end{aligned}$$

## 3.3. *Linearity*

The completion procedure attempts to generate *all* consequences of a given set of rules. For the purposes of computation, as opposed to theorem proving, far fewer overlappings are needed. In general, to execute a rewrite

[6] The answer predicate on the right-hand side serves to store the result by keeping track of substitutions as they are applied to the $\bar{z}$; it is akin to the "answer literal" of (Green, 1969).

program we only want to apply the completion procedure *linearly*. That is, the goal rule and rules derived from it are only overlapped with the rules of the program. Derived rules are not overlapped with themselves, nor are program rules overlapped with each other. Moreover, derived rules are not used for simplification (i.e., steps (3) and (5) of the procedure are omitted). Alternatively, one may wish to compute in a *forward* direction from a given set of facts, overlapping facts with program rules to generate new facts, but not overlapping facts with facts or rules with rules.

The computation sequence for *div* illustrated above, proceeds linearly from goal to answer. An example of a "forward reasoning" program, proceeding from facts to conclusion, is the following for binary-search:

*Forward Binary Search*

$$\begin{array}{rcl} bin(p, 1) & \rightarrow & answer(p) \\ bin(a, n) & \rightarrow & true \\ bin(p, y) \wedge x < f(p + y \div 2) & \rightarrow & bin(p, y \div 2) \\ bin(p, y) \wedge x \geqslant f(p + y \div 2) & \rightarrow & bin(p + y \div 2, (y+1) \div 2) \end{array}$$

where $y \div 2$ is the integer part of $y/2$. Given a monotonic function $f$ and value $x$, this program searches for a position $p$ among the $n$ positions $a$, $a+1, ..., a+n-1$, such that $f(p) \leqslant x < f(p+1)$. Here, $x$, $a$, and $n$ are constants; to use the program, their input values, as well as a system for computing $f$, must be provided.[7] The computation then proceeds by forward-deduction from those givens. In addition to the above rewrite program, we need rules for conjunctions, $u \wedge true \rightarrow u$ and $true \wedge u \rightarrow u$, as well as systems for comparison of values and for addition and division of integers.

To illustrate computation with this program, consider a search between 0 and 15 for the square-root of 16. We input the following rules:

$$\begin{array}{rcl} x & \rightarrow & 16 \\ a & \rightarrow & 0 \\ n & \rightarrow & 16 \\ f(u) & \rightarrow & u^2, \end{array}$$

[7] An alternative would be to have additional arguments to *bin* for the input data $x, a$, and $n$.

along with a program for squaring. These goal rules result in the computation

$$\begin{aligned}
bin(0, 16) &\rightarrow true\\
bin(0, 8) &\rightarrow true\\
bin(4, 4) &\rightarrow true\\
bin(4, 2) &\rightarrow true\\
answer(4) &\rightarrow true.
\end{aligned}$$

That is, the square-root of 16 is (within one of) 4.[8]

Completion, when applied to rules that represent clauses, acts like full resolution (see Hsiang and Dershowitz, 1983). Linear completion, when applied in a backward manner to equations representing Horn-clauses, mimics "linear input" resolution (Kolwalski, 1979), but also allows one to do more. Besides resolving clauses, rules are applied to simplify terms; that way, equality between terms can be treated equationally.

Any Horn-clause may be directly translated into a single rewrite rule; the converse is not the case. The Horn-clause (in Prolog syntax)

$$A :- B, C,$$

meaning $B \wedge C \supset A$, corresponds to the rule

$$A \wedge B \wedge C \quad \rightarrow \quad B \wedge C$$

(i.e., $A \wedge B \wedge C = B \wedge C$, which is the same as $B \wedge C \supset A$). A rule of the form

$$A \wedge B \quad \rightarrow \quad B \wedge C$$

is stronger than the above Horn-clause and means that $B \supset (A \equiv C)$. A rule

$$A \quad \rightarrow \quad B \wedge C$$

is even stronger; it has $A$ true *if and only if* $B$ and $C$ hold. (For forward reasoning, this rule would be oriented $B \wedge C \rightarrow A$ instead, to enable the fact $A$ to be deduced from facts $B$ and $C$.) Assertions

$$A :-$$

[8] The ordering used in completion only has *true* less than other terms. So a critical pair like $true \wedge 16 < f(6) = bin(4, 2)$ will not be pursued further.

correspond to rules

$$A \quad \rightarrow \quad \text{true}$$

with *true* for right-hand sides; goals

$$:\!-\, B, C,$$

to rules

$$B \wedge C \quad \rightarrow \quad \text{false}$$

with *false* for right-hand sides. To keep track of the substitution, we use $answer(\bar{x})$ in place of *false.*

For systems containing only Horn-clauses and goal rules, linear completion can generate any solution. If a goal

$$A[\bar{x}] \quad \rightarrow \quad answer(\bar{x})$$

overlaps a Horn-clause rule of the form

$$A[\bar{y}] \wedge B[\bar{y}] \wedge C[\bar{y}] \quad \rightarrow \quad B[\bar{y}] \wedge C[\bar{y}]$$

with unifying substitution $\bar{\sigma}$, it produces a subgoal of the form

$$B[\bar{\sigma}] \wedge C[\bar{\sigma}] \wedge answer(\bar{\sigma}) \rightarrow B[\bar{\sigma}] \wedge C[\bar{\sigma}].$$

(Note that there would be no point using this rule to solve goals of the form $B$ or $C$.) Since $\wedge$ is associative–commutative, such a subgoal acts like

$$B[\bar{\sigma}] \wedge C[\bar{\sigma}] \wedge answer(\bar{\sigma}) \wedge \alpha \quad \rightarrow \quad B[\bar{\sigma}] \wedge C[\bar{\sigma}] \wedge \alpha$$

and overlaps with rules for $B$ of the form

$$B \wedge D \wedge E \quad \rightarrow \quad D \wedge E.$$

This process continues until all subgoals become *true*, leaving a rule of the form

$$true \wedge \cdots \wedge true \wedge answer(\bar{t}) \quad \rightarrow \quad true \wedge \cdots \wedge true.$$

With the simplifier

$$true \wedge u \quad \rightarrow \quad u,$$

that would simplify to the answer rule

$$answer(\bar{t}) \quad \rightarrow \quad true.$$

## 3.4. *Narrowing*

The *narrowing procedure* is slightly more restricted than linear completion, since rules are used to overlap goals, but not vice-versa. *Narrowing* (Slagle, 1974) is the process of looking for an instance of a term that makes a rule applicable (not solely within the substitution part) and then applying that rule. That is, suppose a term $t[\bar{y}] = u[s][\bar{y}]$ contains a (not necessarily proper) nonvariable subterm $s$ that can be unified with a left-hand side $l$ of a rule $l[\bar{x}] \rightarrow r[\bar{x}]$ by most general substitution $\bar{\sigma}$ (renaming the variables $\bar{x}$ and $\bar{y}$ so that they are disjoint), i.e., $s[\bar{\sigma}] = l[\bar{\sigma}]$. Then that substitution $\sigma$ is first applied to $t$—yielding $u[s][\bar{\sigma}] = u[l][\bar{\sigma}]$—and then that rule $l \rightarrow r$ is applied—yielding $u[r][\bar{\sigma}]$. If $u[r][\bar{\sigma}]$ is not in normal form, then it is reduced further, say to $\hat{t}$. The term $t$ is then said to *narrow* to $\hat{t}$. Fay (1979) showed how narrowing may be used to solve equations, or, in other words, to unify two terms up to equality in a given theory (see also Lankford and Ballantyne, 1979; Hullot, 1980). To compute, one needs to solve equations of the form $p[\bar{s}, \bar{z}] = true$ (assuming the equations embodied in the program). As pointed out in (Dershowitz, 1982b), the completion procedure can simulate narrowing.[9] Given a rule of the form $t \rightarrow c$ (where $c$ is smaller than other terms), linear completion will generate the new rule $\hat{t} \rightarrow c$, without aborting, whenever a left-hand side can be unified with a subterm of $t$. "Narrowing" (to solve equations) is combined with Horn-clause resolution in Eqlog (Goguen and Meseguer, 1984).

To compute, the *narrowing procedure* is initially given a program $R$ and a set $E$ containing the single rule

$$g[z_1, \ldots, z_n] \qquad \rightarrow \qquad answer(z_1, \ldots, z_n)$$

for goal $g$ and output variables $z_1, \ldots, z_n$. The only restriction placed on $R$ is that the constant *true* be irreducible. As soon as $E$ contains a subgoal $answer(\bar{t}) \rightarrow true$, the procedure terminates successfully with answers $\bar{t}$. And if $answer(\bar{t}[z]) \rightarrow z$ is generated (for some boolean variable $z$), then the answers $\bar{t}[true]$ are returned. If $E$ contains no such answer rule, some goal $g[\bar{z}] \rightarrow answer(t_1[\bar{z}], \ldots, t_n[\bar{z}])$ is removed. (If ever no subgoals remain in $E$, the procedure terminates with failure.) All subgoals $g' \rightarrow answer(t'_1, \ldots, t'_n)$ obtainable from $g$ by a rule $l \rightarrow r$ in $R$ are added to $E$, where $g'$ is a normal form of $g[\bar{\sigma}]$ (after first being reduced by $l \rightarrow r$) and $\bar{\sigma}$ is a most general substitution for $\bar{z}$ such that a nonvariable subterm of $g$ unifies with $l$. The partial answers $t'_i$ are normal forms of $t_i[\bar{\sigma}]$. If $g' = true$, then $answer(t'_1, \ldots, t'_n) \rightarrow true$ is added instead.

As before, we require that the procedure execute fairly, i.e., no subgoal

[9] Narrowing can also be simulated in Prolog, but only by decomposing terms. See Plaisted and Greenbaum, 1984; Tamaki, 1984.

placed in $E$ that can be narrowed by some rule is ignored forever. The following theorem provides a sufficient condition for the narrowing procedure to compute with a given rewrite system:

THEOREM 10. *The narrowing procedure computes an input/output relation* $p[\bar{x}, \bar{z}]$, *when given a rewrite program* $R$, *if* $R$ *is sound* (*with respect to some theory* $E$), *terminates for all ground terms equal to true* (*in* $E$), *and is correct with respect to ground input terms* $p[\bar{s}, \bar{t}]$ *equal to true and the constant true. This, provided that the procedure executes fairly.*

What this theorem means is that if the rewrite system evaluates a ground term of the form $p[\bar{s}, \bar{t}]$ to *true*, whenever it is true, then starting the narrowing procedure off with a goal rule $p[\bar{s}, \bar{z}] \rightarrow \mathit{answer}(\bar{z})$, where $\bar{s}$ are the input values and $\bar{z}$ are variables, will eventually generate a subgoal of the form $\mathit{answer}(\bar{t}) \rightarrow \mathit{true}$ such that $p[\bar{s}, \bar{t}]$ is in fact true (if such a $\bar{t}$ exists for the given $\bar{s}$).

*Proof.* Analogous to (Lankford, 1975; Hullot, 1980), we show that if a term $g[\bar{t}]$ reduces to *true* (and only to *true*) under $R$, then narrowing will generate $\mathit{answer}(\bar{t}') = \mathit{true}$, when given the program $R$ and subgoal $g[\bar{z}] \rightarrow \mathit{answer}(\bar{t}[\bar{z}])$. Here $\bar{t}'$ is a normal form of $\bar{t}[\bar{\sigma}]$ and $\bar{\sigma}$ is a substitution for $\bar{z}$ that is at least as general as $\bar{t}$. In particular, then, a satisfiable goal $p[\bar{s}, \bar{z}]$ will generate an answer rule. The proof is by induction on the smallest ground instance $g[\bar{\sigma}]$ of $g[\bar{z}]$ that reduces to *true* and only to *true* (smallest, with respect to the derivation relation $\overset{+}{\Rightarrow}$ restricted to true ground terms, which is well founded). Note that for $g[\bar{\sigma}]$ to be smallest, $\bar{\sigma}$ itself must be irreducible.

If $g[\bar{\sigma}]$ is irreducible, but true, then $g$ must be the constant *true* (or a variable $z$) and the procedure terminates successfully. If $g[\bar{\sigma}]$ is reducible (while $\bar{\sigma}$ is not), then it must be reducible by applying a rule $l \rightarrow r$ such that $g[\bar{z}]$ is overlapped by $l$, say via the most general unifier $\bar{\mu}$, where $\bar{\mu}$ is at least as general as $\bar{\sigma}$, (i.e., $\bar{\sigma} = \bar{\mu}[\bar{v}]$ for some substitution $\bar{v}$). Let $g'$ be a normal form of $g[\bar{\mu}]$ (obtained by first applying $l \rightarrow r$). In that case, the narrowing procedure will generate the equation $g' = \mathit{answer}(\bar{t}')$ from the goal rule $g[\bar{z}] \rightarrow \mathit{answer}(\bar{t}[\bar{z}])$. But $g[\bar{\sigma}] = g[\bar{\mu}[\bar{v}]] \overset{+}{\Rightarrow} g'[\bar{v}]$ and $g'[\bar{v}]$ reduces to *true* and only to *true* since $g[\bar{\sigma}]$ does. The desired result then follows from the induction hypothesis applied to $g'$, since its instance $g'[\bar{v}]$ is smaller than $g[\bar{\sigma}]$. ∎

In particular, if $R$ is a canonical system for a theory $E$ defining $p$, and the constant *true* is irreducible under $R$, then the system must be correct with respect to true ground terms. But $R$ need not be confluent (cf. Goguen and

Meseguer, 1984). If $R$ is sound for $E$, one can use the correctness methods of Section 2.2 to show that all true ground terms are reducible to *true*.[10]

Thus, narrowing is a "complete" computation method for programs that reduce true ground terms (true, with respect to $R$ considered as equations) to the term *true*, in the sense that narrowing is guaranteed to generate an answer satisfying any satisfiable goal. Narrowing suffices to compute with the division program, since the system reduces all ground terms of the form $div(a, b, q, r)$ to *true* whenever the numbers $q$ and $r$ are the quotient and remainder, respectively, of the number $a$ and (the nonzero) $b$. Though the system is confluent, it is not canonical for the theory of integer quotient and remainder, since it does not include rules to reduce false instances to *false*. To see that all true ground terms reduce to *true*, note that the two rules cover the two cases $a \geqslant b$ and $a < b$, and that (as the system is sound) $q$ and $r$ are functionally dependent on $a$ and $b$.

With rewrite systems, negation can be handled in a purely equational manner, by including rules for false cases. Since rules are equivalences, negated goals $\neg A$ can be solved by narrowing $A$ to *false*. For example, the following program for *append* solves both true and false cases:

*Complete List Append*

| | | |
|---|---|---|
| $append(x \cdot X, Y, z \cdot Z)$ | $\to$ | $x = z \wedge append(X, Y, Z)$ |
| $append(nil, y \cdot Y, z \cdot Z)$ | $\to$ | $y = z \wedge append(nil, Y, Z)$ |
| $append(nil, nil, z \cdot Z)$ | $\to$ | *false* |
| $append(x \cdot X, Y, nil)$ | $\to$ | *false* |
| $append(X, y \cdot Y, nil)$ | $\to$ | *false* |
| $append(nil, Y, Y)$ | $\to$ | *true* |

These six rules are correct with respect to *all* ground terms of the form $append(L, M, N)$ and the constants *true* and *false*. They require an additional program for $=$ (one that returns *true* or *false* given ground elements), along with propositional rules for conjunctions.

[10] To *narrow* with Horn-clause programs, one must consider all goal rules to be extended with a new variable. That is, each subgoal $A \to answer(\bar{t})$ is actually $A \wedge \alpha \to answer(\bar{t}) \wedge \alpha$. Such an extended subgoal is narrowed to $B' \wedge C' \wedge \beta \to answer(\bar{t}') \wedge \beta$ by an extended rule $A \wedge B \wedge C \wedge \beta \to B \wedge C \wedge \beta$.

## 4. Synthesis

In this section, we illustrate how the full completion procedure itself may be used to reason about rewrite programs. In particular we show that it can synthesize a program from (equational) specifications. Other deductive approaches to program synthesis include (Burstall and Darlington, 1977; Clark, 1981; Hogger, 1981; Manna and Waldinger, 1980; Murray, 1982).

The simplest use of completion for program generation is to develop a full program from an incomplete one. Take, as an example, the division program. As pointed out in the previous section, that program requires the use of a unification algorithm that takes + to be associative and commutative and to have identity element 0. To eliminate the need for handling 0 within unification, the two rules

$$\begin{aligned} div(x+y+1, y+1, q+1, r) &\quad\to\quad div(x, y+1, q, r) \\ div(x, x+z+1, 0, x) &\quad\to\quad true \end{aligned}$$

can be overlapped with the additional rule

$$x+0 \quad\to\quad x.$$

Overlapping with the first rule adds seven rules (for the cases: $x=0$, $y=0$, $q=0$, $x=y=0$, $x=q=0$, $y=q=0$, $x=y=q=0$); overlapping with the second adds three more ($x=0$, $z=0$, $x=z=0$). For example, letting $x=0$ in the second rule gives the special case

$$div(0, z+1, 0, 0) \quad\to\quad true.$$

The program derived in this manner uses "only" associative–commutative unification.

As an example of how completion may be used to specialize a program, consider the binary-search program of the previous section, and suppose we wish to apply it to the computation of square roots. That suggests completing the program together with the following rules

$$\begin{aligned} f(u) &\quad\to\quad u^2 \\ a &\quad\to\quad 0 \\ n &\quad\to\quad x+1. \end{aligned}$$

The first rule says that we are inverting the squaring function; the others, that the square root of the natural number $x$ is within the range 0 to $x$. The resultant program is

| *Integer Square Root* | | |
|---|---|---|
| $bin(p, 1)$ | $\rightarrow$ | $answer(p)$ |
| $bin(0, x+1)$ | $\rightarrow$ | $true$ |
| $bin(p, y) \wedge x < (p+y \div 2)^2$ | $\rightarrow$ | $bin(p, y \div 2)$ |
| $bin(p, y) \wedge x \geqslant (p+y \div 2)^2$ | $\rightarrow$ | $bin(p+y \div 2, (y+1) \div 2)$ |

More generally, we are interested in generating a program from specifications. Suppose that we wish to synthesize a program for some predicate $p[\bar{x}, \bar{z}]$, given an axiomatization $E'$ of the problem domain and a set $E''$ of equations specifying the required properties of $p$. We can start the completion procedure off with $E = E' \cup E''$ (or with some of the axioms as rules) and run it until a program $R$ is generated that computes the specification $p$. The monotonic well-founded ordering supplied to the completion procedure should ensure that terms containing "specification" symbols are greater than corresponding terms containing the defined goal symbol, which in turn should be greater than (the minimal term) *true*. The particular choice of ordering will, of course, affect the program derived. Note that this approach requires that a program be specified *equationally*; that in itself may require auxiliary definitions (cf. the need for definitions of specification symbols in program verification, Boyer and Moore, 1979).

Given an appropriate ordering, the completion procedure will find a program meeting the specifications, *unless* it aborts.

THEOREM 11. *If there exists a terminating rewrite system R that is sound for a set of equations E and correct with respect to ground terms equal to true and the constant true, then the completion procedure, given E and a monotonic well-founded ordering under which the rules of R are reductions, will generate such a program. This, provided that the procedure executes fairly and does not abort.*

The following proof is analogous to the proof of "uniqueness" of confluent rewrite systems (see Metivier, 1983; Lankford and Ballantyne, 1983; Dershowitz and Marcus, 1984). That completion may abort even when given a reduction ordering for $R$ is shown in (Dershowitz and Marcus, 1984).

*Proof.* We show that completion will generate a system $R'$ such that all true (in $E$) ground terms reduce to *true*. (By Theorem 10, such a system can always be used as a program.) Let $l \to r$ be a rule in $R$. The equation $l = r$ must be valid in $E$. By Theorem 9, then, completion will eventually generate enough rules for $l$ and $r$ to reduce under $R'$ to the same term $u$. Now, it cannot be that $l = u$, since that would mean that $r$ reduces to $l$ in $R'$, contradicting the fact that $l \succ r$ under the well-founded ordering supplied to the completion procedure. It follows that any term reducible by $R$ is also reducible by $R'$.

The remainder of the proof is by induction with respect to $\succ$. Let $t$ be a true ground term. If $t = true$, then we are done. If not, then $t$ must be reducible by $R$, and therefore must reduce by $R'$ to some $u$ $(\prec t)$. Since $u$ is also true (in $E$) and is smaller than $t$ (with respect to $\succ$), by the induction hypothesis, $u$—and $t$ too—must reduce to *true*. ▌

Returning to our division example, suppose that the following definition of multiplication of natural numbers is given:

| *Multiplication* | | |
|---|---|---|
| $m \times (n+1)$ | $\to$ | $m \times n + m$ |
| $m \times 0$ | $\to$ | $0$ |

where $+$ and $\times$ are associative and commutative (with identity elements, 0 and 1, respectively). In addition, suppose we have available all necessary facts about propositions and inequalities (the use of which we will point out as we go along). Given the specification

$$r \leqslant y \wedge x = (y+1) \times q + r \quad \to \quad div(x, y+1, q, r)$$

of integer division, and the recursive path ordering (with the function symbols in decreasing order: $\wedge$, $=$, $<$, *div*, $+$, 1, 0, *true*), the completion procedure generates the following synthesis steps:

By overlapping the specification with $m \times 0 \to 0$ (unifying $y+1$ with $m$ and $q$ with 0) and simplifying $0 + r \Rightarrow r$, we get

$$r \leqslant y \wedge x = r \quad \to \quad div(x, y+1, 0, r). \tag{1}$$

By overlapping (1) with the fact $u = u \to true$ (unifying $x$, $u$, and $r$) and simplifying $r < y \wedge true \Rightarrow r < y$, we get

$$x \leqslant y \quad \to \quad div(x, y+1, 0, x). \tag{2}$$

Next, by overlapping (2) with the fact $u \leqslant u+z \rightarrow$ true (unifying $u$ with $x$ and $y$ with $x+z$), we obtain

$$div(x,\ x+z+1,\ 0,\ x) \quad \rightarrow \quad true. \tag{3}$$

Now, by using (2) to reduce $r \leqslant y$ in the specification to $div(r,\ y+1, 0, r)$ (and multiplying out using the first rule for multiplication), we get

$$div(r,\ y+1,\ 0,\ r) \wedge x = y \times q + q + r \quad \rightarrow \quad div(x,\ y+1,\ q,\ r). \tag{4}$$

Overlapping (4) with $m \times (n+1) \rightarrow m \times n + m$ (letting $q = n+1$) yields

$$div(r,\ y+1, 0, r) \wedge x = y \times q + y + q + 1 + r \quad \rightarrow \quad div(x,\ y+1, q+1, r). \tag{5}$$

Finally, by overlapping (5) with the fact $u + w = v + w \rightarrow u = v$ (letting $w = y+1$), we obtain

$$div(x+y+1,\ y+1, q+1, r) \quad \rightarrow \quad div(x,\ y+1, q, r). \tag{6}$$

The two rules, (3) and (6), are the program of the previous section.

Note that, in general, specifications are themselves executable by the *full* completion procedure. That is, given the facts, specification, and goal rule, the same answer will be generated, albeit with considerably more effort.

## 4.1. *Another Example*

In the following example, we synthesize the forward-reasoning rewrite program of the previous section. That program searches for an integral position $p$ such that the input value $x$ lies between $f(p)$ and $f(p+1)$, for monotonically nondecreasing function $f$. To start the completion procedure off, it is given the output specification

$$x \geqslant f(p) \wedge x < f(p+1) \quad \rightarrow \quad answer(p) \tag{1}$$

($x$ should lie between $f(p)$ and $f(p+1)$), input specification

$$f(u+v) \geqslant f(u) \quad \rightarrow \quad true \tag{2}$$

$$x \geqslant f(a) \quad \rightarrow \quad true \tag{3}$$

$$x < f(a+n) \quad \rightarrow \quad true \tag{4}$$

($f$ is monotonically nondecreasing and $x$ lies between $f(a)$ and $f(a+n)$), and two facts about transitivity of inequality:[11]

$$u<v \wedge w\geqslant v \wedge u<w \quad \rightarrow \quad u<v \wedge w\geqslant v \tag{5}$$

$$v\geqslant u \wedge w\geqslant v \wedge w\geqslant u \quad \rightarrow \quad v\geqslant u \wedge w\geqslant v. \tag{6}$$

The synthesis requires the programmer to introduce a generalization *bin*$(p, v)$ of *answer*$(p)$, and to introduce halving of integers, to guide the synthesis to binary search, rather than linear search. (The ordering on symbols has *bin* between the specification symbols and the goal symbol *answer*.)

The first step is a requisite generalization of (1) on the part of the programmer:

$$x\geqslant f(p) \wedge x<f(p+y) \quad \rightarrow \quad bin(p, y), \tag{7}$$

replacing 1 with $y$ on the left-hand side. This generates the rule

$$bin(p, 1) \quad \rightarrow \quad answer(p), \tag{8}$$

in place of (1), as well as

$$x<f(a+y) \quad \rightarrow \quad bin(a, y) \tag{9}$$

$$bin(a, n) \quad \rightarrow \quad true \tag{10}$$

by overlapping with (3) and (4). Transitivity (5, 6), together with (2), generate

$$w<f(u) \wedge w<f(u+v) \quad \rightarrow \quad w<f(u) \tag{11}$$

$$w\geqslant f(u) \wedge w\geqslant f(u+v) \quad \rightarrow \quad w\geqslant f(u+v). \tag{12}$$

In turn, these and (7) generate

$$bin(p, v+w) \wedge x<f(p+v) \quad \rightarrow \quad bin(p, v) \tag{13}$$

$$bin(p, v+w) \wedge x\geqslant f(p+v) \quad \rightarrow \quad bin(p+v, w). \tag{14}$$

[11] Recall that an implication of the form $u \supset v$ is expressed by a rule $u \wedge v \rightarrow u$.

Together, rules (8, 10, 13, 14) give the nondeterministic search program:

*Nondeterministic Search*

$$\begin{aligned}
bin(p, 1) &\rightarrow answer(p)\\
bin(a, n) &\rightarrow true\\
bin(p, v+w) \wedge x<f(p+v) &\rightarrow bin(p, v)\\
bin(p, v+w) \wedge x\geqslant f(p+v) &\rightarrow bin(p+v, w)
\end{aligned}$$

To derive a *binary* search program, we introduce a definition of halving:

$$(u\div 2)+(u+1)\div 2 \quad\rightarrow\quad u. \tag{15}$$

This generates

$$bin(p, y) \wedge x<f(p+y\div 2) \quad\rightarrow\quad bin(p, y\div 2) \tag{16}$$

$$bin(p, y) \wedge x\geqslant f(p+y\div 2) \quad\rightarrow\quad bin(p+y\div 2, (y+1)\div 2) \tag{17}$$

from (13) and (14). Rules (8, 10, 16, 17) are the binary-search program presented in the previous section.

To generate a backward-reasoning version of binary-search, we need only add the definition

$$bin(p, y) \supset answer(z) \quad\rightarrow\quad pos(z, p, y)$$

and complete using the propositional calculus system. Without going into details, that generates

*Backward Binary Search*

$$\begin{aligned}
pos(z, z, 1) &\rightarrow true\\
pos(z, p, y) \wedge x<f(p+y\div 2) &\rightarrow x<f(p+y\div 2) \wedge pos(z, p, y\div 2)\\
pos(z, p, y) \wedge x\geqslant f(p+y\div 2) &\rightarrow x\geqslant f(p+y\div 2)\\
&\qquad \wedge pos(z, p+y\div 2, (y+1)\div 2)
\end{aligned}$$

Given values for $x$, $a$, and $n$ and programs for $f$, $+$, and $\div$, this program will generate an answer by linear completion, reasoning backwards from a goal

$$pos(z, a, n) \quad\rightarrow\quad answer(z).$$

## 5. Discussion

We have illustrated how rewrite rules can be used for general-purpose computation. Each rule is an equality between terms or equivalence between formulas. The result was a nondeterministic programming language that has the advantages of logic programs, including clean syntax, well-understood semantics, and the ability to use the same language (and not just Horn-clauses) for both specification and computation. Rewrite programs have the additional advantage of allowing the direct incorporation of rules for equalities between terms.

We have also seen how the completion procedure may be used to synthesize rewrite programs from higher-level specifications. Completion can, by the same token, be used for the simpler tasks of extending ("compiling") an incomplete program and verifying the correctness of a program with respect to specifications.

It is important to bear in mind the distinction between theorem proving and computing. For synthesis, the completion procedure is used as a theorem prover, to search for solutions to equations. As an equational-programming language-interpreter, however, full completion is too expensive, just as full resolution is not used for Horn-clause programming. For that reason, we restricted the generation of critical pairs to a linear strategy, making computation more directed. But there is always tradeoff between efficiency and "adequacy" of the interpreter. For a language to call itself "logic-based," it is reasonable to insist that any output whose correctness follows logically from the (declarative) interpretation of the program statements should be computable. We have accordingly given (syntactical and semantic) conditions which guarantee that the interpreter will correctly compute the output.

A potential advantage of using rewrite rules for logic-based programming is the ease with which pattern-directed functional (applicative) programs can be incorporated. Another advantage of programming with rewrite rules is the ability to explicitly include false cases and handle negation. A number of languages (including Bellia, Degano, and Levi, 1982; Chester, 1980; Komorowski, 1982; Robinson and Sibert, 1982) provide both logical and functional programming capabilities by combining features of the two. Some other logic languages that go beyond Horn-clauses are Barbuti *et al.* (1985); Bowen (1982); Fribourg (1985); Goguen and Meseguer (1984); Hansson, Haridi and Tärnlund (1982); Malachi, Manna, and Waldinger (1984); Reddy (1985); Subrahmanyam and You (1984).

One problem with the computation scheme described here is the limited "control" a programmer has over the order in which subgoals are attempted. The "procedural interpretation" of Prolog (as opposed to

"Horn-clause programs") gives the programmer control over the order in which statements are applied and subgoals attempted. Some control is also possible with rewrite rules, but with loss of transparency. For example, to force the same left-to-right order of evaluation subgoals as in the Prolog statement

$$A :- B, C,$$

one can use the pair of rewrite rules:

$$\begin{aligned} A \wedge c(B, \bar{x}) &\quad\rightarrow\quad c(B, \bar{x}) \\ c(true, \bar{x}) &\quad\rightarrow\quad C, \end{aligned}$$

where $\bar{x}$ are the variables in $C$ and $c$ is a new function symbol that generates the subgoal $C$ only *after* $B$ is narrowed to *true*. The intent of $c(\alpha, \bar{x})$, then, is $\alpha \wedge C$. (Cf. the methods of simulating conditional evaluation in Brand, Darringer, and Joyner, 1978; Bergstra and Klop; 1982; top-down rewrite rule systems are discussed in Baeten, Bergstra, and Klop, 1984.)

We are currently looking at the possibility of using conditional equations, and "conditional narrowing," for computing (Dershowitz and Plaisted, 1985). With conditionals, control is more transparent, and functional programs are even easier to express. An implementation of rewrite-system computation methods—within REVE (Lescanne, 1983)—is underway (see Dershowitz and Josephson, 1984; see also Rety *et al.*, 1985).

## ACKNOWLEDGMENTS

I sincerely thank Leo Bachmair, Steve Crocker, Jieh Hsiang, Alan Josephson, Yves Kodratoff, Emmanuel Kounalis, Leo Marcus, David Plaisted, Jean-Luc Rémy, and Ehud Shapiro for many stimulating discussions, and the referees for their valuable suggestions.

RECEIVED: November 29, 1982; ACCEPTED: February 11, 1985

## REFERENCES

BACHMAIR, L., AND DERSHOWITZ, N. (1985), Commutation, transformation, and termination, unpublished report, Department of Computer Science, University of Illinois, Urbana, August.

BACHMAIR, L., AND PLAISTED, D. A. (1985), Termination orderings for associative commutative rewriting systems, *J. Symbolic Comput.* **1**, No. 4, December.

BAETEN, J. C. M., BERGSTRA, J. A., AND KLOP, J. W. (1984), "Priority Rewrite Systems," Report CS-R-8407, Math. Centrum, Amsterdam.

BARBUTI, R., BELLIA, M., LEVI, G., AND MARTELLI, M. (1985), LEAF: A language which integrates logic, equations and functions, *in* "Functional and Logic Programming" (D. deGroot and G. Lindstrom, Eds.), Prentice-Hall, Englewood Cliffs, N.J.

Bellia, M., Degano, P., and Levi, G. (1982), Call by name semantics of a clause language with functions, *in* "Logic Programming" (K. L. Clark and S-Å. Tärnlund, Eds.), Academic Press, New York.

Bergstra, J. A., and Klop, J. W. (1982), "Conditional Rewrite Rules: Confluence and Termination," Report IW *198/82* MEI, Math. centrum, Amsterdam.

Birkhoff, G. (1935), On the structure of abstract algebras, *Proc. Cambridge Philos. Soc.* **31**, 433–454.

Bowen, K. A. (1982), Programming with full first-order logic, *in* "Machine Intelligence 10" (P. Hayes, D. Michie, and Y-H. Pao, Eds.), pp. 421–440, Horwood, Chichester, W. Sussex, U.K.

Boyer, R. S., and Moore, J. S. (1979), "A Computational Logic," Academic Press, New York.

Brand, D., Darringer, J. A., and Joyner, W. J., Jr. (1978), "Completeness of Conditional Reductions," Report RC 7404, IBM Research Center, Yorktown Heights, N.Y., December.

Burstall, R. M., and Darlington, J. (1977), A transformation system for developing recursive programs, *J. Assoc. Comput. Mach.* **24**, No. 1, 44–67.

Burstall, R. M., Mac Queen, D. B., and Sannella, D. T. (1980), HOPE: An experimental applicative language, *in* "Conference Record of the 1980 LISP Conference," Stanford, Calif., pp. 136–143.

Chester, D. (1980), HCPRVR: An interpreter for logic programs, *in* "Proceedings of the First Annual National Conference on Artificial Intelligence," Stanford, Calif.

Clark, K. L. (1981), "The Synthesis and Verification of Logic Programs," Research Report DOC 81/36, Department of Computing, Imperial College, London, England, September.

Clocksin, W., and Mellish, C. (1981), "Programming in Prolog," Springer, New York.

Curry, H. B., and Feys, R. (1958), "Combinatory Logic," North-Holland, Amsterdam.

Dershowitz, N. (1982; 1979), Orderings for term-rewriting systems, *J. Theoret. Comput. Sci.* **17**, No. 3, 279–301; *in* "Proceedings of the Symposium on Foundations of Computer Science," San Juan, P. R., October, pp. 123–131.

Dershowitz, N. (1986), Applications of the Knuth–Bendix completion procedure, *in* "Proceedings of the Seminaire d'Informatique Theorique," Paris, France, December.

Dershowitz, N. (1985a), Termination, *in* "Proceedings of the International Conference on Rewriting Techniques and Applications," Dijon, France, May.

Dershowitz, N. (1985b), Synthesis by completion, *in* "Proceedings of the Ninth International Joint Conference on Artificial Intelligence," Los Angeles, Calif., August, pp. 208–214.

Dershowitz, N., Hsiang, J., Josephson, N., and Plaisted, D. A. (1983), Associative-commutative rewriting, *in* "Proceedings of the Eighth International Joint Conference on Artificial Intelligence," Karlsruhe, West Germany, August 1983, pp. 940–944.

Dershowitz, N., and Josephson, N. A. (1984), Logic programming by completion, *in* "Proceedings of the Second International Conference on Logic Programming," Uppsala, Sweden, July, pp. 313–320.

Dershowitz, N., and Marcus, L. (1984), "Existence and Construction of Rewrite Systems," Office of Information Sciences Research, The Aerospace Corp., El Segundo, Calif., revised August.

Dershowitz, N., and Plaisted, D. A. (1985), Logic programming *cum* applicative programming, *in* "Proceedings, 1985 Symposium on Logic Programming," Boston, Mass., July, pp. 54–66.

Fages, F. (1983), "Formes canoniques dans les algèbres booléennes, et application à la démonstration automatique en logique de premier ordre," Thèse, Université de Paris VI, Paris, France, June.

Fages, F. (1984), Associative-commutative unification, *in* "Proceedings of the Seventh International Conference on Automated Deduction," Napa, Calif., May, pp. 194–208.

FAY, M. (1979), First-order unification in an equational theory, *in* "Proceedings of the Fourth Workshop on Automated Deduction," Austin, Texas, pp. 161–167.

FRIBOURG, L. (1985), SLOG: A logic programming language interpreter based on clausal superposition and rewriting, *in* "Proceedings of the 1985 Symposium on Logic Programming," Boston, Mass., July, pp. 172–184.

FUTATSUGI, K. GOGUEN, J. A., JOUANNAUD, J. P., AND MESEGUER, J. (1985), Principles of OBJ2, *in* "Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages," New Orleans, La., January, pp. 52–66.

GERHART, S. L., MUSSER, D. R., THOMPSON, D. H., BAKER, D. A., BATES, R. L., ERICKSON, R. W., LONDON, R. L., TAYLOR, D. G., AND WILE, D. S. (1980), An overview of AFFIRM: A specification and verification system, *in* "Proceedings of IFIP Congress 80," Tokyo, Japan, October, pp. 343–347.

GOGUEN, J. A. (1980), How to prove algebraic inductive hypotheses without induction, *in* "Proceedings Fifth Conference on Automated Deduction," Les Arcs, France, July, pp. 356–373.

GOGUEN, J. A., AND MESEGUER, J. (1984), Equality, types, modules and (why not?) generics for logic programming, *J. Logic Programming* **1**, No. 2, pp. 179–210.

GRÄTZER, G. H. (Ed.) (1969), "Universal Algebra," 2nd ed., Van Nostrand, Princeton, N. J.

GREEN, C. C. (1969), "The Application of Theorem-proving to Question-answering," Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, Calif.

HANSSON, A., HARIDI, S., AND TÄRNLUND, S-Å. (1982), Properties of a logic programming language, *in* "Logic Programming" (K. L. Clark and S-Å. Tärnlund, Eds.), pp. 267–280, Academic Press, London.

HEARN, A. C. (1971), REDUCE 2—A system for algebraic manipulation, *in* "Proceedings of the Second Symposium on Symbolic and Algebraic Manipulation," Los Angeles, Calif.

HOGGER, C. J. (1981), Derivation of logic programs, *J. Assoc. Comput. Mach.* **28**, No. 2, 372–392.

HSIANG, J. (1982), "Topics in Automated Theorem Proving and Program Generation," Ph.D. thesis, Report R-82-1113, Department of Computer Science, University of Illinois, Urbana, Ill., December.

HSIANG, J., AND DERSHOWITZ, N. (1983), Rewrite methods for clausal and non-clausal theorem proving, *in* "Proceedings of the Tenth EATCS International Colloquium on Automata, Languages and Programming," Barcelona, Spain, July, pp. 331–346.

HUET, G. (1981), A complete proof of correctness of the Knuth–Bendix completion algorithm, *J. Comput. Systems Sci.* **23**, No. 1, 11–21.

HUET, G., AND HULLOT, J. M. (1980), Proofs by induction in equational theories with constructors, *in* "21st Annual Symposium on Foundations of Computer Science," Lake Placid, N.Y., pp. 96–107.

HUET, G., AND LANKFORD, D. S. (1978), "On the Uniform Halting Problem for Term Rewriting Systems," Rapport Laboria 283, IRIA, Le Chesnay, France, March.

HUET, G., AND OPPEN, D. C. (1980), Equations and rewrite rules: A survey, *in* "Formal Language Theory: Perspectives and Open Problems" (R. Book, Ed.), pp. 349–405, Academic Press, New York.

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

HULLOT, J. M. (1980b), "Compilation de formes canoniques dans les théories équationnelles," Thèse, Université de Paris-Sud, Orsay, France, November.

JOUANNAUD, J. P., AND KIRCHNER, H. (1984), Completion of a set of rules modulo a set of equations, *in* "Proceedings of the Eleventh Symposium on Principles of Programming Languages," Salt Lake City, Utah, January.

JOUANNAUD, J. P., KIRCHNER, C., AND KIRCHNER, H. (1983), Incremental construction of unification algorithms in equational theories, *in* "Proceedings of the Tenth EATCS Internat. Colloq. Automata, Languages, and Programming," Barcelona, Spain, July, pp. 361–373.

Jouannaud, J. P., and Muñoz, M. (1984), Termination of a set of rules modulo a set of equations, *in* "Proceedings of the 7th Internat. Conference on Automated Deduction," Napa, Calif., May, pp. 175–193.

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

Kirchner, H. (1984), A general inductive completion algorithm and application to abstract data types, *in* "Proceedings of the Seventh International Conference on Automated Deduction," Napa, Calif., May, pp. 282–302.

Knuth, D. E. (1968), "Fundamental Algorithms," Addison–Wesley, Reading, Mass.

Knuth, D. E., and Bendix, P. B. (1970), Simple word problems in universal algebras, *in* "Computational Problems in Abstract Algebra," pp. 263–297, Pergamon, Oxford, U.K.

Komorowski, H. J. (1982), QLOG—The programming environment for PROLOG in LISP, *in* "Logic Programming" (K. L. Clark and S-Å Tärnlund, Eds.), pp. 315–322, Academic Press, New York.

Kounalis, E., and Zhang, H. (1985), A general completeness test for equational specifications, unpublished report, Centre de Recherche en Informatique de Nancy, Nancy, France.

Kowalski, R. A. (1974), Predicate logic as programming language, *in* "Proceedings of the IFIP Congress," Amsterdam, The Netherlands, pp. 569–574.

Kowalski, R. A. (1979), "Logic for Problem Solving," North-Holland, Amsterdam, 1979.

Lankford, D. S. (1975), "Canonical Algebraic Simplification in Computational Logic," Memo ATP-25, Automatic Theorem Proving Project, University of Texas, Austin, Texas, May.

Lankford, D. S. (1981), "A Simple Explanation of Inductionless Induction," Memo MTP-14, Department of Mathematics, Louisiana Tech. University, Ruston, La., August.

Lankford, D. S., and Ballantyne, A. M. (1977a), "Decision Procedures for Simple Equational Theories with Commutative Axioms: Complete Sets of Commutative Reductions," Memo ATP-35, Department of Mathematics and Computer Sciences, University of Texas, Austin, Texas, March.

Lankford, D. S., and Ballantyne, A. M. (1977b), "Decision Procedures for Simple Equational Theories with Commutative-Associative Axioms: Complete Sets of Commutative–Associative Reductions," Memo ATP-39, Department of Mathematics and Computer Sciences, University of Texas, Austin, Texas, August.

Lankford, D. S., and Ballantyne, A. M. (1979), The refutation completeness of blocked permutative narrowing and resolution, *in* "Proceedings of the Fourth Workshop on Automated Deduction," Austin, Texas, pp. 53–59.

Lankford, D. S., and Ballantyne, A. M. (1983), On the uniqueness of term rewriting systems, unpublished note, Department of Mathematics, Louisiana Tech. University, Ruston, La., December.

Lescanne, P. (1983), Computer experiments with the REVE term rewriting system generator, *in* "Proceedings of the Tenth Symposium on Principles of Programming Languages," Austin, Texas, January, pp. 99–108.

Livesey, M., and Siekmann, J. (1976), "Unification of $A + C$-terms (Bags) and $A + C + I$-terms (Sets)," Intern. Ber. Nr. 5/76, Inst. fur Informatik, University Karlsruhe, Karlsruhe, West Germany.

Malachi, Y., Manna, Z., and Waldinger, R. J. (1984), TABLOG: The deductive tableau programming language, *in* "Proceedings of the ACM Lisp and Functional Programming Conference," pp. 323–330.

Manna, Z. (1974), "Mathematical Theory of Computation," McGraw–Hill, New York.

MANNA, Z., AND NESS, S. (1970), On the termination of Markov algorithms, *in* "Proceedings of the Third Hawaii International Conference on System Science," Honolulu, Hawaii, January, pp. 789–792.

MANNA, Z., AND WALDINGER, R. J. (1980), A deductive approach to program synthesis, *ACM Trans. Programming Lang. Systems* **2**, No. 1, 90–121.

METIVIER, Y. (1983), About the rewriting systems produced by the Knuth–Bendix completion algorithm, *Inform. Process. Lett.* **16**, 31–34.

MURRAY, N. V. (1982), Completely non-clausal theorem proving, *Artif. Intell.* **18**, No. 1, 67–85.

MUSSER, D. R. (1980), On proving inductive properties of abstract data types, *in* "Proceedings, 7th ACM Symposium on Principles of Programming Languages," Las Vegas, Nev., January, pp. 154–162.

PADAWITZ, P. (1983), "Correctness, Completeness, and Consistency of Equational Data Type Specifications," Bericht Nr. 83–15, Technische Universität, Berlin, West Germany.

PETERSON, G. E., AND STICKEL, M. E. (1981), Complete sets of reductions for some equational theories, *J. Assoc. Comput. Mach.* **28**, No. 2, 233–264.

PLAISTED, D. A. (1980), Partial correctness and semantic confluence of term-rewriting systems, unpublished manuscript, Department of Computer Science, University of Illinois, Urbana, Ill.

PLAISTED, D. A. (1983; 1984), An associative path ordering, *in* "Proceedings of an NSF Workshop on the Rewrite Rule Laboratory," Schenectady, N.Y. September, pp. 123–136; Report 84GEN008, General Electric Research and Development, April.

PLAISTED, D. A. (1985), Semantic confluence tests and completion methods, *Inform. Contr.* **65**.

PLAISTED, D. A., AND GREENBAUM, S. (1984), Problem representations for backchaining and equality in resolution theorem proving, *in* "Proceedings First Annual AI Applications Conference," Denver, Colo., December, pp. 417–423.

REDDY, U. (1985), Narrowing as the operational semantics of functional languages, *in* "Proceedings, 1985 Sympos. on Logic Programming," Boston, Mass., July, pp. 138–151.

RETY, P., KIRCHNER, C., KIRCHNER, H., AND LESCANNE, P. (1985), NARROWER: A new algorithm for unification and its application to logic programing, *in* "Proceedings, First International Conference on Rewriting Techniques and Applications," Dijon, France, May.

ROBINSON, J. A., AND SIBERT, E. E. (1982), LOGLISP: an alternative to PROLOG *in* "Machine Intelligence 10" (P. Hayes, D. Michie, and Y-H. Pao, Eds.), pp. 399–419, Horwood, Chichester, W. Sussex, U.K.

SLAGLE, J. R. (1974), Automated theorem-proving for theories with simplifiers, commutativity, and associativity, *J. Assoc. Comput. Mach.* **21**, No. 4, 622–642.

STICKEL, M. E. (1981), A unification algorithm for associative-commutative functions, *J. Assoc. Comput. Mach.* **28**, No. 3, 423–434.

SUBRAHMANYAM, P. A., AND YOU, J. H. (1984), FUNLOG = functions + logic: A computational model integrating functional and logic programming, *in* "Proceedings of the IEEE International Symposium on Logic Programming, Atlantic City, N.J., pp. 144–153.

TAMAKI, H. (1984), Semantics of a logic programming language with a reducibility predicate, *in* "Proceedings of the IEEE International Symposium on Logic Programming," Atlantic City, N.J., February, pp. 259–264.

THIEL, J. J. (1984), Stop losing sleep over incomplete data type specifications, *in* "Proceedings of the Eleventh Symposium on Principles of Programming Languages," Salt Lake City, Utah.

WALDINGER, R. J. (1969), "Constructing Programs Automatically Using Theorem Proving," Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., May.

WALDINGER, R. J., AND LEVITT, K. N. (1974), Reasoning about programs, *Artif. Intell.* **5**, No. 3, 235–316.

WATTS, D. E., AND COHEN, J. K. (1980), Computer implemented set theory, *Amer. Math. Monthly* **87**, No. 7, 557–560.