J. Symbolic Computation (1993) **11**, 1–000

Deductive and Inductive Synthesis of Equational Programs†

NACHUM DERSHOWITZ AND UDAY S. REDDY

Department of Computer Science, University of Illinois at Urbana-Champaign, U.S.A.

(Received 5 June 1993)

An equational approach to the synthesis of functional and logic program is taken. In this context, the synthesis task involves finding executable equations such that the given specification holds in their standard model. Hence, to synthesize such programs, induction is necessary. We formulate procedures for inductive proof, as well as for program synthesis, using the framework of "ordered rewriting". We also propose heuristics for generalizing from a sequence of equational consequences. These heuristics handle cases where the deductive process alone is inadequate for coming up with a program.

1. Introduction

In seminal work, Burstall and Darlington (1977) showed how functional programs, expressed as equations, can be transformed to more efficient ones using equational reasoning. Given a specification of a new function to be synthesized, they use the original program equations forward ("unfolding") and backward ("folding") in a controlled fashion and obtain a recursive program for the new function. The method has come to be called the "fold-unfold" method and forms an important component in reasoning about functional programs. (See (Bird and Wadler, 1988).) Significant effort has been devoted to building automated systems based on the methodology (see, for example, (Darlington, 1981; Feather, 1982)), which has been adapted to reasoning about logic programs (Hogger, 1976; Tamaki and Sato, 1984; Deville, 1990). Partial evaluation systems, increasingly successful in recent times (Bjorner *et al.*, 1988; ACM, 1991), are also based on the fold-unfold method.

In building reliable general-purpose program synthesis systems, however, several issues arise:

> How does one determine if the transformed programs are correct? (While the soundness is immediate from the technique, termination and completeness remain concerns.)

† First author's research supported in part by the U. S. National Science Foundation under Grants CCR-90-07195 and CCR-90-24271 and by a Meyerhoff Visiting Professorship at the Weizmann Institute of Science. The second author's research was supported in part by U. S. National Science Foundation grant CCR-87-00988, NASA grant NAG-1-613 and a grant from Motorola Corporation.

0747–7171/90/000000 + 00 $03.00/0 © 1993 Academic Press Limited

How does one control the application of equations? (Naïve application of equations leads to large search spaces. The controlled application used by Burstall and Darlington is sometimes restrictive.)

How does the method generalize to forms of programs (and logics) other than equational ones (such as conditional equations, Horn clauses or first-order clauses)?

What role does (mathematical) induction play in the synthesis process?

How does the method relate to other methodologies of deductive synthesis, like (Manna and Waldinger, 1980; Bibel and Hörnig, 1984; Smith, 1985)?

In attempting to answer some of these questions, we are led to the framework of *term rewriting*, the best known technique of controlled equational reasoning. Term rewriting was first used in automated reasoning by Knuth and Bendix (1970) for solving word problems in equational theories. Two fundamental operations underlie the technique: *rewriting* and *superposition*. Rewriting uses a terminating system of oriented equations, called *(rewrite) rules*, to rewrite a term to a "normal form". Superposition uses existing rewrite rules to deduce a new equation. The combination of the two techniques achieves extremely high performance in equational reasoning. In recent work, term rewriting techniques have been extended to deal with unoriented equations (Hsiang and Rusinowitch, 1987; Bachmair *et al.*, 1989; Martin and Nipkow, 1990; Peterson, 1990; Bachmair and Dershowitz, to appear), conditional equations (Bergstra and Klop, 1986; Kaplan, 1987; Kounalis and Rusinowitch, 1987; Ganzinger, 1991), and first-order reasoning (Hsiang and Dershowitz, 1983; Hsiang and Rusinowitch, 1991; Zhang and Kapur, 1988; Bronsard and Reddy, 1991; Nieuwenhuis and Orejas, 1991). See (Huet and Oppen, 1980; Dershowitz, 1989; Dershowitz and Jouannaud, 1990; Klop, 1992) for accessible surveys of this rapidly developing area.

The contributions of this paper are threefold: First, we enrich the basic equational reasoning techniques used by Burstall and Darlington with additional structure to obtain rewrite-based reasoning. Second, we propose (mathematical) induction techniques to define and ensure the correctness of synthesized programs. Third, we demonstrate how inductive generalization techniques supplement the basic deductive techniques to achieve an automated program synthesis system. This paper consolidates and extends our previous work reported in (Dershowitz, 1982; Dershowitz, 1985b; Reddy, 1989; Reddy, 1990b; Dershowitz and Pinchover, 1990). In the cited work, we treated rewrite systems; here, we generalize those techniques to a mix of oriented and unoriented equations, using the notion of "ordered rewriting". This makes the method complete for the class of deductively verifiable programs. The application of ordered rewriting to program synthesis or inductive proofs has also been considered in (Bachmair, 1988; Gramlich, 1989; Bellegarde, 1991); recent work of Goldammer (1992) is also based on similar ideas. Franhöfer and Furbach (1986) compare rewriting techniques with the plain equational methods of Burstall and Darlington. In early work, Kapur and Srivas (1985) present many ideas closely related to those here.

We begin with an overview of program synthesis. It is followed, in Section 3, by a description of the basic properties of equational programs. Sections 4 and 5, respectively, deal with deductive and inductive reasoning. Details of the formal methods for synthesis are given in Section 6. Section 7 describes the heuristic techniques used in conjunction with the formal ones. We conclude with a brief discussion.

2. Overview

Suppose we wish to synthesize a program for some function f and are given a specification S for f, together with an axiomatization E of the problem domain. There are two ways to think about the program synthesis process: We can try to generate all interesting logical consequences of S and E in the hope of eventually obtaining some set of equations which serves as a program for f. Or, we can try to reduce the specification S to simpler equations, using E, in the hope of eventually obtaining equations simple enough to serve as a program for f. The former, *forward reasoning*, approach seems to underlie Burstall and Darlington (1977), whereas the latter, *backward reasoning*, approach is the basis of Manna and Waldinger (1980). Interestingly—in the context of equational reasoning—the two approaches produce very similar results and one can view the same set of deductions from both the forward and backward reasoning points of view.

For example, consider the following axiomatization of append and reverse functions for lists:

$$\begin{aligned} append(x, nil) &= x && (2.1)\\ append(nil, y) &= y && (2.2)\\ append(w \cdot u, y) &= w \cdot append(u, y) && (2.3)\\ append(append(x, y), z) &= append(x, append(y, z)) && (2.4)\\ reverse(nil) &= nil && (2.5)\\ reverse(w \cdot u) &= append(reverse(u), w \cdot nil) && (2.6) \end{aligned}$$

Suppose we want a program for the function *revap* which reverses its first argument and appends it to the second argument. It is specified by the equation:

$$revap(x, y) = append(reverse(x), y) \qquad (2.7)$$

To synthesize a program from this specification, we first note that the subterm $reverse(x)$ can be simplified using the defining equations (2.5,2.6) of *reverse* if x is instantiated to nil and $w \cdot u$, respectively. It is then fairly straightforward to derive the following equations:

$$\begin{aligned} revap(nil, y) &= append(reverse(nil), y) && \text{from (2.7)}\\ &= append(nil, y) && \text{by (2.5)}\\ &= y && \text{by (2.2)}\\ revap(w \cdot u, y) &= append(reverse(w \cdot u), y) && \text{from (2.7)}\\ &= append(append(reverse(u), w \cdot nil), y) && \text{by (2.6)}\\ &= append(reverse(u), append(w \cdot nil, y)) && \text{by (2.4)}\\ &= append(reverse(u), w \cdot append(nil, y)) && \text{by (2.3)}\\ &= append(reverse(u), w \cdot y) && \text{by (2.2)}\\ &= revap(u, w \cdot y) && \text{by (2.7)} \end{aligned}$$

All the steps use axioms to replace "equals by equals", except for the last step which uses the original specification (2.7) for a smaller instance. (Such use of the original specification is termed "folding" in (Burstall and Darlington, 1977).) The two equations derived above form a program for *revap*:

$$\begin{aligned} revap(nil, y) &= y\\ revap(w \cdot u, y) &= revap(u, w \cdot y) \end{aligned} \qquad (2.8)$$

which is similar to what one would write in a pattern-directed functional programming language like ML (Paulson, 1991).

The above calculations can be viewed as a forward reasoning process. The two derived equations are evidently logical consequences of the specification (2.7). However, it is not entirely clear that the two equations form a correct (terminating and complete) program for *revap*. If we design a synthesis procedure based on the forward reasoning approach, we would have to use some other mechanism to ensure the correctness of the derived program. Moreover, we would also need some heuristic guidance to navigate through the space of all logical consequences so that "interesting" consequences are found.

We can also view the above calculations as a backward reasoning process in which the specification (2.7) acts as the theorem being proved, the program (2.8) consists of the axioms necessary to prove the theorem, and the synthesis process itself provides a backward proof of the theorem. Note that all the equational replacement steps are equivalence-preserving. Thus, they can be viewed as either forward or backward steps. The initial instantiation step is justified by noting that, to prove (2.7) as an inductive theorem, it *is* adequate to prove the two instances. The final folding step is justified as the use of (2.7) as the inductive hypothesis applied to the smaller instance $x = u$, when proving that the hypothesis holds for the larger instance $x = w \cdot u$.

There are good reasons to prefer the backward reasoning view to the forward reasoning view. For one, it eliminates the need for navigating through all possible logical consequences in search of the program, giving better control over the search process. For another, it integrates inductive reasoning with the deductive process, so that the derived programs are guaranteed to be correct. Therefore, we adopt the backward reasoning view in the rest of the paper.

The synthesis of the *revap* function is just part of the more general task of finding an efficient program for *reverse*. The program represented by the axioms (2.5–2.6) takes time quadratic in the length of the list. To find an efficient program, we must eliminate its use of *append*. However, unlike the above synthesis, this cannot be achieved by deduction alone. No amount of replacing equals by equals will eliminate the use of *append*. To successfully synthesize a program, we need an "insight" (a "*eureka*" step, as Burstall and Darlington termed it). We must recognize that we need an auxiliary function to compute the quantity $append(reverse(u), v)$, where the extra variable v has been introduced to hold the partial result of reversal. Having synthesized a program for it (the *revap* function), we can use its specification (2.7) to simplify the program of *reverse* as follows:

$$\begin{array}{rcl} reverse(nil) & = & nil \\ reverse(w \cdot u) & = & revap(u, w \cdot nil) \end{array} \tag{2.9}$$

and, thereby, eliminate the use of *append*.

Can an automatic synthesis procedure find the *eureka* step? Indeed, a number of heuristics can be used to postulate auxiliary functions (similar to postulating lemmas in inductive proofs). For the problem on hand, a simple generalization heuristic (Boyer and Moore, 1977; Arsac and Kodratoff, 1982) suffices. We first attempt to perform a derivation starting from the specification

$$reverse(w \cdot u) \quad = \quad append(reverse(u), w \cdot nil)$$

in the same manner as that of the *revap* function above. We notice that the subterm $reverse(u)$ can be simplified using the defining equations of *reverse* if u is instantiated

Table 1. Equational axiomatization of propositional calculus

$$
\begin{array}{rcl}
\neg u & = & u \Leftrightarrow \mathbf{false} \\
u \supset v & = & (u \vee v) \Leftrightarrow v \\
u \wedge v & = & (u \vee v) \Leftrightarrow v \Leftrightarrow u \\
u \vee u & = & u \\
u \vee \mathbf{true} & = & \mathbf{true} \\
u \vee \mathbf{false} & = & u \\
u \Leftrightarrow u & = & \mathbf{true} \\
u \Leftrightarrow \mathbf{true} & = & u \\
(u \Leftrightarrow v) \vee w & = & (u \vee w) \Leftrightarrow (v \vee w) \\
u \Leftrightarrow v & = & v \Leftrightarrow u \\
(u \Leftrightarrow v) \Leftrightarrow w & = & u \Leftrightarrow (v \Leftrightarrow w) \\
u \vee v & = & v \vee u \\
(u \vee v) \vee w & = & u \vee (v \vee w)
\end{array}
$$

to nil and $w' \cdot u'$. This gives the equations:

$$
\begin{array}{rcl}
reverse(w \cdot nil) & = & append(reverse(nil), w \cdot nil) \\
& = & append(nil, w \cdot nil) \\
& = & w \cdot nil
\end{array}
$$

$$
\begin{array}{rcl}
reverse(w \cdot w' \cdot u') & = & append(append(reverse(u'), w' \cdot nil), w \cdot nil) \\
& = & append(reverse(u'), append(w' \cdot nil, w \cdot nil)) \\
& = & append(reverse(u'), w' \cdot w \cdot nil)
\end{array}
$$

At this point, a successful derivation would be able to apply a folding step. Since we are unable to do this, we attempt to find a more general specification expression which might enable a folding step. The expression $append(reverse(u), v)$ generalizes the original specification expression as well as the current one. (In fact, the least generalization $append(reverse(u), w \cdot v)$ works as well. This is what our algorithm would find.) This gives the auxiliary function needed to complete the synthesis.

Many program synthesis tasks involve conditional reasoning in addition to equational reasoning. Though term rewriting techniques have been extended to conditional equations, as well as first-order clauses, we do not get into these technical areas in this paper. Instead, we will use an equational axiomatization of Boolean algebras. Here, "=" denotes equality of truth values, that is, logical equivalence. All predicate symbols are treated as function symbols and so are the logical connectives $\neg$, $\wedge$, $\vee$, $\supset$, and $\Leftrightarrow$. Table 1 gives an equational axiomatization of propositional calculus in this notation (compare (Hsiang and Dershowitz, 1983)).

Consider the following axiomatization of addition, subtraction, multiplication and equality of natural numbers in successor notation, wherein the number n is represented as $s^n(0)$:

$$
\begin{array}{rcll}
x + 0 & = & x & (2.10) \\
x + y & = & y + x & (2.11) \\
(x + y) + z & = & x + (y + z) & (2.12) \\
x \times 0 & = & 0 & (2.13) \\
x \times s(y) & = & (x \times y) + x & (2.14)
\end{array}
$$

$$x \approx x \quad = \quad \mathbf{true} \tag{2.15}$$
$$x \approx y + z \quad = \quad x - z \approx y \tag{2.16}$$

Here, "$\approx$" is the equality comparison operator for naturals. Notice that we only specify equations for the *true* case of $\approx$. This is because we want to view these equations as logic programs where the false cases simply "fail". See (Dershowitz, 1985a) for a discussion of how logic programs are treated in the equational framework.

Suppose our goal is to produce a program for natural number division specified by

$$div(x, s(y), q, r) \quad = \quad (r \leq y) \wedge (x \approx s(y) \times q + r) \tag{2.17}$$

The predicate $div(x, s(y), q, r)$, meaning that dividing x by $y + 1$ gives quotient q and remainder r, is specified by stating that the remainder is less than the divisor and that the quotient and remainder are related to the divisor and dividend by the appropriate equation.

As in the *revap* example, we instantiate q to 0 and $s(z)$, so as to simplify the subterm $s(y) \times q$ by axioms (2.13–2.14). (The other possibility is to simplify $r \leq y$, but this choice does not lead to a good program.) The following equations are then obtained:

$$\begin{array}{rcll}
div(x, s(y), 0, r) & = & r \leq y \wedge x \approx s(y) \times 0 + r & \text{from (2.17)} \\
& = & r \leq y \wedge x \approx 0 + r & \text{by (2.13)} \\
& = & r \leq y \wedge x \approx r + 0 & \text{by (2.11)} \\
& = & r \leq y \wedge x \approx r & \text{by (2.10)} \\
div(x, s(y), s(z), r) & = & r \leq y \wedge x \approx s(y) \times s(z) + r & \text{from (2.17)} \\
& = & r \leq y \wedge x \approx (s(y) \times z + s(y)) + r & \text{by (2.14)} \\
& = & r \leq y \wedge x \approx s(y) \times z + (s(y) + r) & \text{by (2.12)} \\
& = & r \leq y \wedge x \approx s(y) \times z + (r + s(y)) & \text{by (2.11)} \\
& = & r \leq y \wedge x \approx (s(y) \times z + r) + s(y) & \text{by (2.12)}
\end{array}$$

For the first case, we can instantiate r to x to make domain fact (2.15) applicable. This gives a more compact version, namely:

$$div(x, s(y), 0, x) \quad = \quad x \leq s(y)$$

For the second case, we can apply axiom (2.16) with the substitution $\{x \mapsto x,\ y \mapsto s(y) \times z + r,\ z \mapsto s(y)\}$. This gives:

$$\begin{array}{rcl}
div(x, s(y), s(z), r) & = & r \leq y \wedge x - s(y) \approx s(y) \times z + r \\
& = & div(x - s(y), s(y), z, r)
\end{array}$$

where the last step is a folding step using the specification. The two equations

$$\begin{array}{rcl}
div(x, s(y), 0, x) & = & x \leq s(y) \\
div(x, s(y), s(z), r) & = & div(x - s(y), s(y), z, r)
\end{array}$$

can be viewed as a logic program for division.

3. Equational Programs

First, we briefly explain our notation. By an *alphabet* of function symbols Σ, we mean a set of function symbols together with an *arity* associated with each symbol. The set of variable-free terms over Σ (respecting arities) is denoted G and they are called *ground*

terms; the set of terms over Σ allowing variables (from some set X) is denoted T and they are called *free terms*, or simply *terms*.

An *equation* is a pair of terms written as $r = s$. Given a set of equations E and terms t and t', $E \models t = t'$ if and only if there are terms $t_0, t_1, \ldots, t_n$ $(n \geq 0)$ such that

$$t \equiv t_0 \leftrightarrow_E t_1 \leftrightarrow_E \ldots \leftrightarrow_E t_n \equiv t'$$

where $\leftrightarrow_E$ is the "replacing equals by equals" relation of E ("$\equiv$" is used to denote syntactic identity). A sequence such as the one exhibited is called an "equational proof". The standard relational notations $\leftrightarrow_E^+$ and $\leftrightarrow_E^*$ are used to denote the transitive and reflexive-transitive closures, respectively, of $\leftrightarrow_E$. Thus, $E \models t = t'$ iff $t \leftrightarrow_E^* t'$.

Equational programs work by replacing equals by equals. However, this cannot be done in an arbitrary fashion; a program must make "progress" in evaluating terms. We specify the notion of progress via a well-founded order $\succ$ with certain extra properties stated in Section 3.1. Let $\succ$ be such an order. We say that t *rewrites* to s if $t \leftrightarrow_E s$ and $t \succ s$. This fact is denoted by writing $t \rightarrow_{E,\succ} s$ but we often omit "$\succ$" and write $t \rightarrow_E s$. The idea is that an equation is used for rewriting only in one direction, the direction that achieves reduction by the order $\succ$. Since $\succ$ is well-founded, every rewrite sequence $t_0 \rightarrow_E t_1 \rightarrow_E \cdots$ is finite and results in an unrewritable term, called a *normal form* (which need not be unique). Equational programs are "executed" by rewriting ground terms to normal forms. Since this form of rewriting always reduces the term it is applied to in the well-founded order $\succ$, execution is always terminating.

An equation $t = s$ is said to have a *rewrite proof* if there are terms $t_0, t_1, \ldots, t_n$ and $s_0, s_1, \ldots s_m$ such that

$$t \equiv t_0 \rightarrow_E t_1 \rightarrow_E \ldots \rightarrow_E t_n \equiv s_m \leftarrow_E \ldots \leftarrow_E s_1 \leftarrow_E s_0 \equiv s$$

where $\leftarrow_E$ is the relational inverse of $\rightarrow_E$. Thus, a rewrite proof is an equational proof that rewrites both t and s to some common normal form. Again, $\rightarrow_E^+$ and $\rightarrow_E^*$ will be used respectively to denote the transitive and reflexive-transitive closures of $\rightarrow_E$.

If $r = s$ is an equation in E such that $r \succ s$, no matter what terms are substituted for the variables of the equation, we may, alternatively, write the equation as $r \rightarrow s$. The idea is that such an equation is always used in one direction: to rewrite instances of r to the corresponding instances of s. The equation $r \rightarrow s$ is often called a *rewrite rule* to emphasize this fact, but note that all our equations are rewrite rules in a more general sense: they are always used for rewriting along a reducing direction although the direction may vary from instance to instance. Conventional term rewriting theory (Knuth and Bendix, 1970; Huet and Oppen, 1980) deals with *rewrite systems*, sets of equations all oriented in a particular direction. The idea that unoriented equations can also be used for rewriting, provided they are used along a reducing direction was developed in (Hsiang and Rusinowitch, 1987; Bachmair *et al.*, 1989). This form of rewriting is now called *ordered rewriting*. The results of this paper generalize our previous results (Dershowitz, 1982; Dershowitz, 1985a; Dershowitz, 1985b; Reddy, 1989; Reddy, 1990b; Dershowitz and Pinchover, 1990) to the framework of ordered rewriting.

The mixing of programs and program synthesis with termination issues requires some explanation. Demanding that the rewrite relation be always included in a well-founded order has two consequences: First, it ensures that programs terminate along *all* evaluation paths. While this is a reasonable requirement for most common programs, some applications also require programs that do not terminate, but make progress indefinitely. Programs in lazy functional languages (Bird and Wadler, 1988) often exhibit this prop-

erty. We envisage that the techniques of this paper will eventually be extended to such programs by suitable relaxation of the termination requirements, or an extension of the operational semantics of rewriting.

A second consequence of the termination of rewrite relations is that the automated reasoning procedures have some heuristic guidance about the direction they should employ in reducing problems. Without such guidance, the reasoning procedures need to explore too many possibilities resulting in large search spaces and much redundancy. It will be seen that the well-founded orders used for the rewrite relations play an essential role in the problem specification for program synthesis as well as in the synthesis process itself.

However, it must be noted that the use of *ordered rewriting* allows us to avoid limitations that have been traditionally caused by termination criteria. First, it allows us to include in programs equations that are not orientable as rewrite rules. Commutativity equations such as (2.11) are well-known examples of this. Since such equations are symmetric, orienting them in either direction results in infinite rewrite sequences and, so, they cannot be included in conventional programming languages based on rewriting. On the other hand, ordered rewriting allows us to use such equations in programs and rewrites terms of the form $t+s$ to $s+t$ whenever $t+s \succ s+t$. If we use a $\succ$ ordering that is total on ground terms then, for any ground terms t and s, either $t+s$ is rewritable to $s+t$ or *vice versa*. So, we have rewrite proofs for all such equalities even though the equation itself is not orientable as a rewrite rule. Secondly, it is often hard (or impossible) to design well-founded orders that cover all the equations that may be derived in a deduction procedure. Procedures based on traditional rewriting *fail* when they encounter equations that cannot be oriented as rewrite rules. Our program synthesis procedure avoids this problem by employing equations and ordered rewriting rather than rewrite rules.

3.1. Orderings

We now state the required properties of the well-founded order. A well-founded order $>$ on ground terms is called a *complete reduction order* if (a) it is total on ground terms, (b) it has the replacement property ($s > s'$ implies $t[s] > t[s']$), and (c) it has the subterm property ($t > s$ whenever s is a proper subterm of t). (We use the notation $t[\,]$ to denote a "context", that is, a term with a unique hole. The notation $t[s]$ denotes the term with the hole filled by a term s. When necessary, the position of a hole may be made precise, as in $t[s]_p$; the subterm of t at p is denoted $t|_p$.) Such an order must be well-founded (Dershowitz, 1987). A complete reduction order $>$ is extended to a partial order $\succ$ on free terms by defining

$$t \succ s \quad \Longleftrightarrow \quad t\sigma > s\sigma \text{ for all ground substitutions } \sigma$$

Note that $\succ$ inherits the replacement and subterm properties. In addition, it has the substitution property: $t \succ s$ implies $t\theta \succ s\theta$ for all substitutions θ. (In practice, it suffices to approximate this order by using an ordering of free terms that can be extended to a complete reduction order.)

One complete reduction order of particular interest in program synthesis is the *lexicographic path order* (Dershowitz, 1987; Kamin and Lévy, 1980). Assume a total order $>_P$ on function symbols, referred to as a "precedence". Then, the lexicographic path order $>$ is defined inductively by $t \equiv f(t_1, \ldots, t_m) > g(s_1, \ldots, s_n) \equiv s$ iff one of the following conditions holds:

$t_i \geq s$ for some $i = 1, \ldots, m$;
$f >_P g$ in the precedence order and $t > s_i$ for all $i = 1, \ldots, n$;
$f = g$ $(m = n)$, $\langle t_1, \ldots, t_m \rangle$ is greater than $\langle s_1, \ldots, s_m \rangle$ by the (left-to-right) lexicographic extension of $>$ and, in addition, $t > s_i$ for all $i = 1, \ldots, n$.

In practice, one also specifies the sequence in which the arguments of a function symbol must be compared lexicographically (so that one obtains flexibility in ordering the arguments of a function symbol). The extension of this ordering to free terms can be computed (Comon, 1990), or one can approximate it by using the above definition for free terms, as well. (One can also work with a partial precedence, since it can be extended to a complete reduction order.)

We illustrate the path order with examples. Consider the precedence

$$reverse > append > \cdot > nil$$

and equations (2.1–2.6). With the corresponding lexicographic path order, every left-hand side is greater than the corresponding right-hand side. For example, for (2.1), we have

$$append(x, nil) \succ x$$

because x is a subterm of $append(x, nil)$. For equation (2.4),

$$append(append(x, y), z) \succ append(x, append(y, z))$$

because $\langle append(x, y), z \rangle$ is greater than $\langle x, append(y, z) \rangle$ (by lexicographic order and subterm property), and the left-hand side term is greater than x as well as $append(y, z)$ (the last of these by another application of the definition of $\succ$). For equation (2.6),

$$reverse(w \cdot u) \succ append(reverse(u), w \cdot nil)$$

because $reverse > append$ in the precedence order and $reverse(w \cdot u) \succ reverse(u)$ and $reverse(w \cdot u) \succ w \cdot nil$ ($reverse > \cdot$, $reverse > nil$ and $reverse(w \cdot u) \succ w$).

To handle the specification (2.7) of *revap*, we must extend the precedence order to include *revap*. A good heuristic in choosing precedences is that a symbol f should be greater than all the symbols that may be introduced during the evaluation of $f(t_1, \ldots, t_n)$. Since the evaluation of $revap(t, u)$ must not introduce *reverse* and *append*, but may introduce $\cdot$ and *nil*, we choose the order

$$reverse > append > revap > \cdot > nil$$

Since

$$append(reverse(x), y) \succ revap(x, y)$$

in the extended term ordering, the specification (2.7) cannot be used left-to-right in evaluating terms of the form $revap(t, u)$. This defines the problem for the program synthesis procedure: It must find simpler equations which can be used to evaluate $revap(t, u)$.

3.2. Programs

A rewrite relation $\rightarrow_{E,\succ}$ is said to be *confluent* if, whenever $t \leftrightarrow_E^* u$, there is a rewrite proof of $t = u$. It is said to be *ground confluent* if this property holds for all ground terms t and u. We also say that E is confluent or ground confluent (with respect to $\succ$) if these properties hold. Confluence implies that all terms have unique normal forms; ground confluence implies that ground terms have unique normal forms.

DEFINITION 3.1. *An* equational program *is a finite set E of equations, together with a computable complete reduction order $>$, such that $\rightarrow_{E,>}$ is ground confluent.*

The ground confluence requirement means that the results of programs are deterministic.

Ground confluence is not a decidable property (Kapur *et al.*, 1987). On the other hand, confluence of rewrite rules is decidable and forms a *sufficient* condition for ground confluence. So, in practice, we use the following method. We divide equational theories into parts: *axioms* and inductive *theorems*. The axioms serve to define the function symbols and are used in the evaluation of terms. The inductive theorems form additional knowledge about the problem domain which may be used in program synthesis. If the set of axioms is ground confluent, then the full theory with inductive theorems is also ground confluent. (See Section 5.) Ground confluence of axioms can then be ensured by checking confluence. Of the equations (2.1–2.6), two (2.1,2.4) are inductive theorems. The others define the functions *append* and *reverse*. By standard results in rewriting, they form a confluent system (no greater side unifies with a non-variable subterm of a greater side). Hence, the whole system is ground confluent.

An equational program is said to be *complete* with respect to a set of ground *input terms* Φ and a set of ground *output terms* Ψ if the normal form of every t in Φ belongs to Ψ. The output terms are typically formed of constructor symbols, such as *nil* and $\cdot$ in the case of list axioms. Sometimes, we want to model equivalences over constructor terms in which case only a subset of constructor terms may be included in Ψ. For example, considering the axioms (2.10–2.12) for $+$ in the unary number system, Ψ includes 0, 1 and $m+1$ where $m \in \Psi$. All other terms, such as $m+0$, $0+m$, $m+(n+k)$ must be reducible. The set of input terms is often the set of *all* terms, but occasionally we want to model partial functions or partial axiomatizations of functions. For example, the natural number axiom (2.15) only models the true case of comparison. We say that an axiomatization is *total* if its set of input terms Φ includes all terms. Otherwise we call it *partial*.

It is not, in general, possible to specify the sets Φ and Ψ in a mechanically verifiable fashion, but (Dershowitz, 1985a) and others give methods for some important cases.

4. Superposition for Deriving Cases

An important component in the informal synthesis procedure outlined in Section 2 is the instantiation of equations for the various cases of their variables. Two questions to be answered in the formalization of the procedure are how to find instantiations that are useful for synthesis, and how to verify that the chosen instantiations are complete. The informal procedure already gives an indication of the answer to the first question: we should choose instantiations that make further simplifications possible. For example, in the synthesis of *revap*, we chose instantiations that enable simplification by axioms (2.2–2.3). For the second question, the general method we use is to assume that the initial axiomatization is complete (in the sense of Sect. 3.2) and, then, use this assumption to find complete sets of instantiations. These issues are elaborated in the present section.

Consider a specification $t = u$, with t and u in normal form, such as:

$$append(reverse(x), y) \quad = \quad revap(x, y) \tag{4.1}$$

The program synthesis task is to derive enough program equations so that every ground instance of the specification is "covered", that is, every ground instance has an equational

proof. For the sake of argument, assume that we already have the necessary program equations as a part of the domain theory. Let $t\sigma = u\sigma$ be a ground instance of the specification that has some equational proof $t\sigma \leftrightarrow_E \ldots \leftrightarrow_E u\sigma$. Focus on the first step of this proof. There must be a domain equation $l = r$ (or $r = l$) such that $t\sigma$ contains an instance of l (say, at position p), and the equational proof has the form

$$t\sigma \equiv t\sigma[l\tau]_p \quad \leftrightarrow_E \quad t\sigma[r\tau]_p \quad \leftrightarrow_E^* \quad u\sigma \tag{4.2}$$

There exists a most general proof schema of this form whose first step has the above structure:

$$t\theta \equiv t\theta[l\theta]_p \quad \leftrightarrow_E \quad t\theta[r\theta]_p \quad = \quad u\theta \tag{4.3}$$

where $\theta = \mathrm{mgu}(l, t|_p)$ is the most general unifier of l and the subterm of t at p. The equation $t\theta[r\theta]_p = u\theta$ is a new specification equation whose program would be a part of the overall program. This equation is called a *paramodulant* of the equations $t = u$ and $l = r$, and the operation deriving it is called *paramodulation* (Robinson and Wos, 1969; Brand, 1975). For example, paramodulating the *revap* specification at the subterm $reverse(x)$ with equations (2.5) and (2.6), we can derive the paramodulants:

$$\begin{array}{rcl} append(nil, y) & = & revap(nil, y) \\ append(append(reverse(w), u), y) & = & revap(w \cdot u, y) \end{array} \tag{4.4}$$

While these two equations suffice to derive a program for *revap*, there are many other paramodulants. For instance, another paramodulant is obtained using (2.4) right to left:

$$\begin{array}{lll} append(reverse(x), append(y, z)) & \leftrightarrow_{(2.4)} & append(append(reverse(x), y), z) \\ & = & revap(x, append(y, z)) \end{array}$$

In fact, there are in all 29 paramodulants using the domain theory (2.1–2.6)!

To cut down this search space, we first note that there is no need to paramodulate into variables (Peterson, 1983). That leaves 5 paramodulants. To cut down further, we use the ideas of ordered rewriting. Since the domain theory E is ground confluent, requiring that $t\sigma = u\sigma$ has an equational proof is equivalent to requiring that it has a *rewrite proof* of the form $t\sigma \rightarrow_E^* v \leftarrow_E^* u\sigma$. This allows us to place the following restrictions on the proof schema (4.2):

1 $t\sigma[l\tau]_p > t\sigma[r\sigma]_p$. The first step must be a rewrite step.
2 $t\sigma > u\sigma$. This results in no loss of generality because $>$ is a well-founded order and the rewrite proof will eventually reduce $t\sigma$ to a term smaller than or equal to $u\sigma$.
3 σ is irreducible. Since $\rightarrow_E$ is terminating, every grounding substitution σ has a normal form σ'. Thus, $t\sigma = u\sigma$ has a rewrite proof if and only if $t\sigma' = u\sigma'$ has a rewrite proof.

Lifting these considerations to the paramodulation proof schema:

$$t\theta[l\theta]_p \quad \leftrightarrow_E \quad t\theta[r\theta]_p \quad = \quad u\theta$$

we can impose the following restrictions:

1 $t\theta[l\theta]_p \not< t\theta[r\theta]_p$, because otherwise $t\sigma[l\tau]_p < t\sigma[r\tau]_p$.
2 $t\theta[l\theta]_p \not< u\theta$, because otherwise $t\sigma[l\tau]_p < u\sigma$.
3 p is a nonvariable position of t, because otherwise we would be reducing σ.

Paramodulants satisfying these restrictions are called *critical pairs.*

DEFINITION 4.1. *A* critical pair *of equations $l = r$ and $t = u$ (whose variables are renamed apart) is an equation*

$$t\theta[r\theta]_p \quad = \quad u\theta$$

where $\theta = \mathrm{mgu}(l, t|_p)$ for a nonvariable position p of t, $t\theta \not\prec t\theta[r\theta]_p$ and $t\theta \not\prec u\theta$. The inference rule of deriving critical pairs is called (ordered) superposition.

The notion of critical pair in (Knuth and Bendix, 1970) is a special case of this where the two equations participating in the inference are rewrite rules $l \to r$ and $t \to u$. In that case, the conditions $t\theta[l\theta]_p \not\prec t\theta[r\theta]_p$ and $t\theta \not\prec u\theta$ are automatically satisfied.

Applying these ideas to the *revap* specification, we find that, of the 29 paramodulants, there are only three critical pairs. These include the two essential critical pairs (4.4) and another one that comes from an overlap with (2.1):

$$append(reverse(x), nil) \quad \to_{(2.1)} \quad reverse(x) \quad = \quad revap(x, nil)$$

While this critical pair, $reverse(x) = revap(x, nil)$, is not necessary for synthesizing a program for *revap*, it is still a useful equation. It can serve as the program for *reverse* instead of the more elaborate program (2.9).

Though superposition is an essential part of our program synthesis procedure, our use of it differs from its conventional usage in completion or refutational theorem proving (Hsiang and Rusinowitch, 1987; Bachmair *et al.*, 1989; Martin and Nipkow, 1990). Conventionally, superposition is a forward inference mechanism used to deduce equational consequences of the given theory. In contrast, we use superposition as a backward inference to reduce a given goal to smaller goals. Another important difference is that, in the conventional framework, superposition is used symmetrically in its two premises. That is, given two equations, either one can be chosen as $l = r$ and the the other used as $t = u$. In contrast, in our case, $t = u$ is always a specification equation and $l = r$ is a domain fact. We overlap a domain fact with a subterm of the specification, but not the other way. Thus, in general, the program synthesis procedure does less work than a completion procedure.

As noted above, some of the critical pairs of a specification equation are necessary for deriving a program. We now address the question which subset of the critical pairs, if any, forms a sufficient set of subgoals. Looking back at the cases (4.4), we can say that these equations form a sufficient set of subgoals because they are obtained from the specification (4.1) using the instantiations $\{x \mapsto nil\}$ and $\{x \mapsto w \cdot u\}$, and these instantiations are "complete". The following definition captures this notion of completeness:

DEFINITION 4.2. *A set of substitutions Θ is said to be* inductively complete *if for every ground substitution σ, there exist θ in Θ and ground substitution τ such that $x\sigma \to_E^* x\theta\tau$ for all variables x. (If the domain theory is partial, then this must hold for all substitutions σ over input terms.)*

For example, using the domain theory (2.1–2.6), the substitutions $\{x \mapsto nil\}$, and $\{x \mapsto w \cdot u\}$ form an inductively complete set because all ground substitutions for x reduce to an instance of one of them.

Notice that it is adequate to restrict attention to irreducible σ's in the above definition, because other substitutions reduce to irreducible ones. We can then simplify the condition $x\sigma \rightarrow_E^* x\theta\tau$ to $x\sigma \equiv x\theta\tau$.

Using this notion, we can define a rule for reasoning by cases as follows:

$$Cases \qquad \frac{t[r_1]\theta_1 = u\theta_1 \quad \ldots \quad t[r_k]\theta_k = u\theta_k}{t[s] = u}$$

if $\{l_i = r_i\}_i \subseteq E$, $\theta_i = \mathrm{mgu}(l_i, s)$, $t[l_i]\theta_i \not< t[r_i]\theta_i$, $t[s]\theta_i \not< u\theta_i$ and $\{\theta_i\}_i$ is inductively complete. That is, given a set of critical pairs of $t[s] = u$ whose overlapping substitutions form an inductively complete set, we can infer the equation itself. The soundness property of the inference is as follows:

LEMMA 4.3. *Given a Cases inference, if all the ground instances of the premises have rewrite proofs in E, then all the ground instances of the conclusion have rewrite proofs in E.*

The *Cases* rule considers superposition at a single position of the given equation. It is also possible to choose any position on either side of the equation for critical pairs, using ideas from (Bachmair, 1988).

DEFINITION 4.4. *A set of equations C is said to be a* cover set *for an equation $t = u$ (with respect to E and $>$) if, for every irreducible ground substitution σ, either $t\sigma \equiv u\sigma$ or there exists an equation $r = s$ (or $s = r$) in C such that $(t\sigma = u\sigma) \rightarrow_E^* (r\tau = s\tau)$ and $\max(t\sigma, u\sigma) > \max(r\tau, s\tau)$ for some substitution τ.*

Here, $\max(t\sigma, u\sigma)$ is the maximum with respect to the complete reduction order $>$. The general rule for *Cases* uses a cover set of $t = u$ as premises.

$$Cases \qquad \frac{r_1 = s_1 \quad \ldots \quad r_k = s_k}{t = u}$$

where $\{r_i = s_i\}_i$ is a cover set of $t = u$. Members of a cover set cannot simply be instances of the conclusion equation; they should incorporate at least one step of reduction in order to satisfy the condition $\max(t\sigma, u\sigma) > \max(r_i\tau, s_i\tau)$. This defines a notion of "progress" for the inference.

To formalize this notion, we define a complexity measure for proofs, as in (Bachmair *et al.*, 1989; Bachmair and Dershowitz, to appear). Consider a ground proof using $E \cup S$ where E is the domain theory and S is some set of equations. We associate with it a complexity measure in $G \cup \{\bot\}$ ordered by an extension of the reduction order where $t > \bot$ for all ground terms t in G. The complexity of a proof step $t \leftrightarrow_E t'$ is $\bot$ and the complexity of $t \leftrightarrow_S t'$ is $\max(t, t')$. The complexity of a proof is the maximum complexity of all its proof steps. So, essentially, the complexity of a proof $t_0 \leftrightarrow \ldots \leftrightarrow t_n$ is the maximum term t_i which participates in an $\leftrightarrow_S$ step and $\bot$ if there is no such term.

LEMMA 4.5. *Given a Cases inference of the above form, for every ground instance $t\sigma = u\sigma$ of the conclusion, there is an equational proof using the premises and the equational system E whose complexity is strictly less than that of $t\sigma \leftrightarrow_{t=u} u\sigma$.*

PROOF. By induction on $\max(t\sigma, u\sigma)$. If σ is a reducible substitution with $\sigma \rightarrow_E^+ \sigma'$, use the inductive hypothesis for $t\sigma' = u\sigma'$. If σ is irreducible, by definition of cover set, there is an equation $r_i = s_i$ such that $(t\sigma = u\sigma) \rightarrow_E^* (r_i\tau = s_i\tau)$ and $\max(t\sigma, u\sigma) > \max(r_i\tau, s_i\tau)$. So, the equation $t\sigma = u\sigma$ has a ground proof of the form

$$t\sigma \rightarrow_E^* r_i\tau \leftrightarrow_{r_i=s_i} s_i\tau \leftarrow_E^* u\sigma$$

whose complexity, $\max(r_i\tau, s_i\tau)$, is less than $\max(t\sigma, u\sigma)$. □

An important question is how to test whether a given set of critical pairs is a cover set. Several methods are possible. A set of terms called *test set* may be computed, such that every irreducible ground term is an instance of some member of the test set (Plaisted, 1985). To check if a given set of critical pairs is a cover set, it is enough to see if each combination of terms from the test set is covered in the overlap substitutions. For instance, for the domain theory (2.1–2.6), $\{nil, w \cdot u\}$ is a test set. This verifies that critical pairs (4.4) form a cover set.

Another method is to use a ground reducibility test. An equation $t = t'$ is said to be *ground reducible* if, for every ground instance $t\sigma = t'\sigma$, either $t\sigma$ is identical to $t'\sigma$ or one of them is reducible. In this case, the set of *all* critical pairs is a cover set. (If one of them is reducible then the larger one is. Suppose $t\sigma$ is the larger term. If it is reducible by some domain equation, then $t\sigma$ is covered by a critical pair between the domain equation and $t = t'$.) The set of all critical pairs is often too large for a cover set. As we have noted, the critical pair $reverse(x) = revap(x, nil)$ need not be in a cover set of the *revap* specification. If extraneous critical pairs are included in a cover set, they might generate other critical pairs and lead to nontermination. (We present an example of this situation in Section 6.) A useful optimization has been suggested in (Fribourg, 1989; Küchlin, 1989). A term is *ground reducible* if every ground instance is reducible. It suffices to consider a subterm s of either t or t' that is ground reducible. Then superposition at the subterm s is enough to obtain a cover set.

Another optimization was suggested in (Kapur *et al.*, 1991): It is enough to restrict attention to irreducible substitutions in the definition of "ground reducible". Whenever $t = s$ is not ground reducible, it equates some pair of irreducible ground terms. An "irreducible ground term" test set can be devised to detect this situation. This form of test set has the property that the equation $t = s$ reduces an irreducible ground term if and only if it reduces some member of the test set. The advantage of this method is that the test set is computed only once and reused in each *Cases* inference. However, this method still requires all critical pairs to be computed for the cover set.

Other methods for testing ground reducibility may found in (Kounalis and Zhang, 1985; Jouannaud and Kounalis, 1989; Bündgen and Küchlin, 1989).

5. Induction

In synthesizing a program from a specification, we must ensure that the derived program satisfies the specification. That is, the specification must be an inductive theorem of the derived program. So, inductive reasoning is an integral part of program synthesis. In this section, we briefly outline our inductive reasoning procedure based on *term rewriting induction*. This method was first presented in (Reddy, 1990b) and is based on the "inductive completion" and "proof by consistency" methods studied in (Musser, 1980;

Huet and Hullot, 1982; Dershowitz, 1982; Dershowitz, 1985a; Kapur and Musser, 1987; Jouannaud and Kounalis, 1989; Fribourg, 1989; Küchlin, 1989; Bachmair, 1988).

An equation e is said to be an *inductive consequence* of an equational system E, written $E \models_{ind} e$, if every ground instance $e\sigma$ follows from E. When E is ground confluent (with respect to $>$), this is equivalent to requiring that $e\sigma$ have a rewrite proof using E. Adding such an inductive theorem to E does not affect its ground confluence. This is one way to build ground confluent equational theories.

The proof of $E \models_{ind} e$ involves three kinds of steps: we can simplify e using the equations in E, we can instantiate it using the *Cases* rule of the previous section, or we can use e as an inductive hypothesis in proving one or more of its cases. Notice that, whenever we use the *Cases* rule, we always reduce the instances $e\sigma$ in complexity. Since simplification and *Cases* always *reduce* the ground instances of the equation the original equation e can be used for simplification of the cases as if it were an "ordinary" equation. This method, sometimes referred to as "inductionless induction", differs from conventional induction in that one never needs to check that the inductive hypothesis is used for a smaller instance than the one being proved. The proof method itself takes care of the condition. Such implicit application of induction may also be found in a variety of program verification methods such as Hoare logic (especially, the treatment of recursion (Hoare, 1971)) and fixed point induction (Manna, 1974; Scott, 1976).

We make these ideas precise by the following inference procedure for pairs of equation sets H and S. We write such a pair as $H \vdash S$; S is a set of conjectures to be proved and H is the set of induction hypotheses which may be assumed in the proof of S. The pair $H \vdash S$ may be read as the judgment "assuming H as induction hypotheses, S", but see Theorem 5.1 for a precise statement. The inference rules of the procedure are as follows:

Axiom	$\dfrac{}{H \vdash \emptyset}$	
Cases	$\dfrac{H \cup \{e\} \vdash S \cup C}{H \vdash S \cup \{e\}}$	if C is a cover set of e
Delete	$\dfrac{H \vdash S}{H \vdash S \cup \{t = t\}}$	
Simplify	$\dfrac{H \vdash S \cup \{e'\}}{H \vdash S \uplus \{e\}}$	if $e \rightarrow_{E \cup H \cup S} e'$
Subsume	$\dfrac{H \vdash S}{H \vdash S \cup \{t[l\theta] = t[r\theta]\}}$	if $l = r$ in H
Hypothesize	$\dfrac{H \vdash S \cup \{e\}}{H \vdash S}$	

The procedure is used by starting with a goal of the form $\emptyset \vdash S_0$ and using some inference rule backwards in each step. If, eventually, a goal of the form $H \vdash \emptyset$ is obtained, the initial theorems in S_0 are all proved and H contains a useful representation of the theorems as well as any lemmas generated in the process. *Simplify* allows a conjecture e to be simplified using equations in E, induction hypotheses in H or *other* conjectures in S. ("$\uplus$" denotes disjoint union.) *Subsume* allows an induction hypothesis to be applied without a concomitant reduction. Note that, in contrast to *Simplify*, this form of an application can be done only once for a conjecture. *Hypothesize* allows one to postulate

new lemmas (*eureka* steps) which may help the proof the theorem. Such lemmas are introduced either by heuristics or by manual intervention.

Consider proving the associativity property of *append* using the rewrite program (2.1–2.3), and suppose the arguments of *append* are compared left to right for the lexicographic path order. We start with the goal:

$$\vdash \{append(append(x,y),z) \quad = \quad append(x,append(y,z))\}$$

Using *Cases*, we can reduce this to

$$\{append(append(x,y),z) \quad \rightarrow \quad append(x,append(y,z))\} \vdash \left\{ \begin{array}{rcl} w \cdot append(y,z) & = & append(nil,append(y,z)), \\ append(w \cdot append(u,y),z) & = & append(w \cdot u,append(y,z)) \end{array} \right\}$$

The first equation simplifies to the identity $append(y,z) = append(y,z)$ and is deleted. The second one simplifies to

$$w \cdot append(append(u,y),z) \quad = \quad w \cdot append(u,append(y,z))$$

Using the inductive hypothesis (by either *Simplify* or *Subsume*), this too reduces to an identity and is deleted. The inductive hypothesis in H is now an inductive theorem and it can be added to the underlying equational theory E as a domain fact.

As another example, assume the following program for *revap*:

$$\begin{array}{rcl} revap(nil,y) & \rightarrow & y \\ revap(w \cdot u,y) & \rightarrow & revap(u,w \cdot y) \end{array}$$

We would like to prove that it satisfies the correctness condition:

$$revap(x,nil) \quad = \quad reverse(x)$$

We start with this as the only conjecture in the goal. However, we immediately notice that we require a more general inductive hypothesis. Hypothesize another conjecture (to be proved as a lemma):

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

(We postpone to Section 7 the issue of how such lemmas may be invented.) Assume that the function symbols are ordered as $revap > reverse > append > \cdot > nil$ in the precedence. We can use *Cases* to reduce the two-equation goal to:

$$\{revap(x,y) \quad \rightarrow \quad append(reverse(x),y)\} \vdash \left\{ \begin{array}{rcl} y & = & append(reverse(nil),y), \\ revap(u,w \cdot y) & = & append(reverse(w \cdot u),y), \\ revap(x,nil) & = & reverse(x) \end{array} \right\}$$

The first equation simplifies to identity and is deleted. The second equation simplifies to

$$\begin{array}{rcl} revap(u,w \cdot y) & = & append(append(reverse(u),w \cdot nil),y) \\ & = & append(reverse(u),w \cdot y) \end{array}$$

The two sides are equal by the inductive hypothesis. Finally, the third equation reduces, using the inductive hypothesis (which is really an inductive "theorem" at this stage), to

$$append(reverse(x),nil) \quad = \quad reverse(x)$$

and this too reduces to identity. The proof is now complete, and we obtain a more general

version of the original equation as a useful rewrite rule to be added to the domain theory of the program.

To prove the soundness of the induction proof procedure, we need to show that all ground instances of the equations in S have proofs using E. The last four inference rules are all instances of the general rule

$$\frac{H \vdash S'}{H \vdash S}$$

where S is provable from $E \cup H \cup S'$ and any proof step $s \leftrightarrow_S t$ using S is more complex than an alternative proof $s \leftrightarrow^*_{E \cup H \cup S'} t$ of the same equation.

THEOREM 5.1. *Let $H \vdash S$ be a derivable judgment. If all ground instances $r\sigma = s\sigma$ of equations in H have proofs using $E \cup S$ of complexity smaller than that of $r\sigma \leftrightarrow_H s\sigma$, then all ground instances of S have proofs using E.*

PROOF. To simplify the argument we introduce some terminology. We say an equation $r = s$ "has (strictly) bounded S-proofs" if every ground instance $r\sigma = s\sigma$ has a proof using $E \cup S$ with complexity (strictly less than) less than or equal to that of $r\sigma \leftrightarrow_{r=s} s\sigma$. (Note that this means, by the replacement property of $<$, that every ground application of $r = s$ of the form $c[r\sigma] = c[s\sigma]$ has a proof with complexity less than or equal to that of $c[r\sigma] \leftrightarrow_{r=s} c[s\sigma]$.) We say $r = s$ "has proofs" if every ground instance $r\sigma = s\sigma$ has a proof using E. So, the statement of the theorem becomes "H has strictly bounded S-proofs $\Longrightarrow$ S has proofs".

The proof is by induction on the derivation of $H \vdash S$. It is trivial for *Axiom*. Suppose

$$\frac{H' \vdash S'}{H \vdash S}$$

is an inference. The plan is to show that the hypothesis of the theorem holds for $H' \vdash S'$ (H' has strictly bounded S'-proofs) whenever it holds for $H \vdash S$ (H has strictly bounded S-proofs) and that the conclusion holds for S (S has proofs) whenever it holds for S' (S' has proofs).

For inferences *Delete* and *Hypothesize*, the proof is trivial. Consider a *Cases* inference

$$\textit{Cases} \qquad \frac{H \cup \{e\} \vdash S \cup C}{H \vdash S \cup \{e\}} \quad \text{where } C \text{ is a cover set for } e$$

Assume that the equations in H have strictly bounded $S \cup \{e\}$-proofs. By Lemma 4.5, e has strictly bounded C-proofs. So, the equations in H (as well as e) have strictly bounded $S \cup C$-proofs. For the conclusion, if the equations in C have proofs, then e has proofs, again, by Lemma 4.5.

Next, consider a *Simplify* inference:

$$\textit{Simplify} \quad \frac{H \vdash S \cup \{t' = s\}}{H \vdash S \cup \{t = s\}} \quad \text{where } t \rightarrow_{E \cup H \cup S} t' \text{ and } t = s \text{ is not in } S$$

Assume that the equations in H have strictly bounded $S \cup \{t = s\}$-proofs. If $t \rightarrow_E t'$, $t = s$ has bounded $\{t' = s\}$-proofs. If $t \rightarrow_H t'$, we show below that $t = s$ has bounded $S \cup \{t' = s\}$-proofs. If $t \rightarrow_S t'$, $t = s$ has bounded $S \cup \{t' = s\}$-proofs. So, in all cases, the equations in H have strictly bounded $S \cup \{t' = s\}$-proofs. For the conclusion, if $S \cup \{t' = s\}$ has proofs then $t = s$ has proofs by essentially the same argument.

To show that $t = s$ has bounded $S \cup \{t' = s\}$-proofs for the case $t \rightarrow_H t'$, consider a ground instance $t\sigma = s\sigma$ and use induction on $\max(t\sigma, s\sigma)$. The instance has a proof $t\sigma \rightarrow_H t'\sigma \leftrightarrow_{t'=s} s\sigma$. For the second step, note that $\max(t'\sigma, s\sigma) \leq \max(t\sigma, s\sigma)$. The first step, by the assumption above, can be replaced by a proof using $S \cup \{t = s\}$ with a complexity strictly less than $\max(t\sigma, t'\sigma) = t\sigma$. If this proof contains a step using $t = s$, say $c[t\tau] \leftrightarrow c[s\tau]$, then $max(c[t\tau], c[s\tau]) < t\sigma \leq max(t\sigma, s\sigma)$. Since $\max(t\tau, s\tau) \leq \max(c[t\tau], c[s\tau])$ by the subterm property of $<$, we can conclude, by induction, that $t\tau = s\tau$ has a proof using $S \cup \{t' = s\}$ of complexity less than or equal to $\max(t\tau, s\tau)$. By replacement property of $<$, $c[t\tau] = c[s\tau]$ has a proof using $S \cup \{t' = s\}$ of complexity less than or equal to $\max(c[t\tau], c[s\tau])$ which is, in turn, strictly less than $\max(t\sigma, s\sigma)$.

Instances of *Subsume* can be verified similarly. □

What if a goal of the form $H \vdash \emptyset$ cannot be obtained? That means that there is an equation $t = s$ in S for which none of the rules *Cases* through *Subsume* are applicable. This means, in particular, that there is no cover set C for $t = s$. We have already seen that if $t = s$ is ground reducible, then the set of all critical pairs with E would be a cover set. So, we conclude that $t = s$ is not ground reducible, that is, there is a ground instance $t\sigma = s\sigma$ such that $t\sigma$ and $s\sigma$ are distinct normal forms by E. Since E is assumed to be ground confluent, $t\sigma = s\sigma$ does not follow from E and, hence, $t = s$ is not an inductive theorem. Thus, whenever an equation $t = s$ cannot be eliminated from S, we have *disproved* the equation. Thus, the induction proof procedure is robust. It fails only if the given conjectures are not inductive theorems. If they are inductive theorems, the procedure may go on indefinitely. Postulating appropriate lemmas using *Hypothesize* will help complete the proof.

6. Program Synthesis

In this section, we return to the problem of program synthesis. To start with, one has a *specification alphabet* Σ, an equational axiomatization E, and a complete reduction order $>$ over G (the ground terms over Σ) such that E is ground confluent. The synthesis problem is specified in terms of a new, *target alphabet* Σ', an equational specification C, and an extension of the reduction order $>$ to G' (the ground terms over $\Sigma \cup \Sigma'$). The reduction order must be extended to Σ' in such a way that, for each new symbol f in Σ', a term containing f is greater than terms constructed from "primitive" operations, and smaller than terms containing specification symbols that may not appear in a program. For example, considering the synthesis problem for *revap*, given by (2.1–2.7), the initial alphabet Σ consists of *reverse*, *append*, $\cdot$ and *nil*, listed in the decreasing order of precedence; the alphabet Σ' consists of *revap* and the precedence order is extended to $reverse > append > revap > \cdot > nil$. This indicates that $\cdot$ and *nil* may appear in the program for *revap*, but not *reverse* or *append*.

The synthesis task is to derive a program P such that (a) P is a *consistent enrichment* of E, that is, not affecting the ground equivalences of G that follow from E, and (b) $E \cup P \models_{ind} S$. We have already seen, in Section 5, how to verify $E \cup P \models_{ind} S$. To infer P, given only E and S, we run the inductive proof procedure with P as an "unknown". The axioms E are fixed, so the goal is to find P, constructed from primitive operations, such that $E \cup P \models_{ind} S$. The equations $t = u$ in S tha tcannot be eliminated by any of the inference rules, called *persisting* equations, require knowledge of P. By accepting the set of all persisting equations of S as P, we can trivially satisfy the requirement

$E \cup P \models_{ind} S$. Of course, not all such P's satisfy the consistent enrichment condition. We return to this issue below.

Consider synthesizing *revap*. The specification is

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

Here, the right-hand side is greater than the left-hand side (because *append* and *reverse* are given higher precedence than *revap*). So, the equation cannot be used as the program for *revap*. Instead, the synthesis procedure must reduce it to simpler equations that have instances of $revap(x, y)$ as the greater side. We consider superposition at the subterm $reverse(x)$, and derive the following cover set by *Cases*:

$$\begin{aligned} revap(nil, y) &= append(nil, y) \\ revap(w \cdot u, y) &= append(append(reverse(u), w \cdot nil), y) \end{aligned}$$

At this stage, we have the specification as an inductive hypothesis in H. The cases simplify to

$$\begin{aligned} revap(nil, y) &= y \\ revap(w \cdot u, y) &= append(reverse(u), w \cdot y) \end{aligned}$$

We can use the inductive hypothesis with *Simplify* to reduce the second right-hand side to $revap(u, w \cdot y)$. (This corresponds to a "folding" step in the terminology of Burstall and Darlington (1977).) No more rules are applicable to these equations. So, the two orientable equations

$$\begin{aligned} revap(nil, y) &\rightarrow y \\ revap(w \cdot u, y) &\rightarrow rev(u, w \cdot y) \end{aligned}$$

form the candidate program for *revap*.

To check for the consistent enrichment condition, we use the following result:

THEOREM 6.1. *Let E and $E \cup P$ be ground confluent sets of equations over alphabets Σ and $\Sigma \cup \Sigma'$, respectively. Then, P is a consistent enrichment of E if, for every ground instance $t\sigma = u\sigma$ of an equation in P such that $t\sigma > u\sigma$, either $t\sigma$ contains target symbols from Σ', or $t\sigma$ and $u\sigma$ are in the specification language G and $t\sigma$ is reducible by E.*

PROOF. We show by induction on $\max(t\sigma, u\sigma)$ that every ground instance $t\sigma = u\sigma$ of an equation in P such that $t\sigma, u\sigma \in G$ has a proof using E. Assume, without loss of generality, that $t\sigma > u\sigma$. By hypothesis, $t\sigma$ is reducible by E. Let $t\sigma \rightarrow_E s$. By ground confluence, the equation $s = u\sigma$ has a rewrite proof using $E \cup P$. All the $\leftrightarrow_P$ steps of this proof necessarily have complexity less than or equal to $\max(t\sigma, u\sigma)$. Since $\max(s, u\sigma) < t\sigma$, we can conclude by induction that, for each P step in the latter proof, there is a proof using E. Hence, $t\sigma = u\sigma$ has a proof using E. □

We use this result as follows: Given a candidate program P_0, we calculate the completion of $E \cup P_0$. (Only the axioms in E need to be used in the completion. Inductive theorems in E do not affect ground confluence.) Suppose completion generates a set $E \cup P_1$. If all equations $t = u$ in P_1 are such that $t > u$ and t contains target symbols, then P_1 is an acceptable program. (It is enough to ensure that for every equation $t = u$ in P_1, $t\sigma > u\sigma$ only if t contains target symbols, for all substitutions σ of specification language ground terms.) If t and u are in the specification language, then we need to verify that $t = u$ is an inductive theorem of E. If t is a specification language term, but u

has target symbols, then $t = u$ is a further specification of Σ' and we continue to derive a program for it.

Thus, synthesis is an iterative process. After finding a candidate program, adding it to the axioms generates certain equational consequences. These consequences may involve problems for further program synthesis. However, we often find that no iteration is needed. For instance, adding the above candidate program for *revap* to the axioms generates no new critical pairs. So, this is indeed the final program for *revap*.

As a somewhat intricate example of the synthesis process, consider the problem of checking two binary trees for the equality of their fringes. (This is a problem considered by Burstall and Darlington (1977).) We start with the following axioms (where *tip* and $\circ$ are constructors for binary trees, f denotes the fringe of a tree and $\approx_L$ is equality comparison for lists):

$$f(tip(x)) = w \cdot nil \tag{6.1}$$
$$f(tip(x) \circ w) = x \cdot f(w) \tag{6.2}$$
$$f((u \circ v) \circ w) = f(u \circ (v \circ w)) \tag{6.3}$$
$$nil \approx_L nil = \mathbf{true} \tag{6.4}$$
$$x \cdot u \approx_L nil = \mathbf{false} \tag{6.5}$$
$$nil \approx_L y \cdot v = \mathbf{false} \tag{6.6}$$
$$x \cdot u \approx_L y \cdot v = x \approx y \wedge u \approx_L v \tag{6.7}$$
$$u \approx_L v = v \approx_L u \tag{6.8}$$

(These are used together with the list axioms (2.1-2.6) and the propositional axioms in Table 1.) The fringe equality of trees is then specified by

$$x \approx_F y = f(x) \approx_L f(y) \tag{6.9}$$

The problem is to synthesize a direct program for $\approx_F$ that does not use f or $\approx_L$. We order the function symbols by the precedence

$$\approx_L > f > \approx_F > \circ > tip > \cdot > nil$$

and have $\circ$ order its arguments from left to right. All the axioms are orientable left to right using this order.

The synthesis proceeds as follows: We can find a cover set for (6.9) by considering superposition at the subterm $f(x)$ on the larger (right) side of the specification. This gives the cases (shown after possible simplification steps):

$$tip(x) \approx_F y = x \cdot nil \approx_L f(y) \tag{6.10}$$
$$tip(x) \circ w \approx_F y = x \cdot f(w) \approx_L f(y) \tag{6.11}$$
$$(u \circ v) \circ w \approx_F y = u \circ (v \circ w) \approx_F y \tag{6.12}$$

The cases (6.10) and (6.11) need further synthesis. This time, we choose $f(y)$ (again on the larger side) for superposition. This gives the cases:

$$tip(x) \approx_F tip(x') = x \approx x' \tag{6.13}$$
$$tip(x) \approx_F tip(x') \circ w' = x \approx x' \wedge nil \approx_L f(w') \tag{6.14}$$
$$tip(x) \approx_F (u' \circ v') \circ w' = tip(x) \approx_F u' \circ (v' \circ w') \tag{6.15}$$
$$tip(x) \circ w \approx_F tip(x') = x \approx x' \wedge f(w) \approx_L nil \tag{6.16}$$

$$tip(x) \circ w \approx_F tip(x') \circ w' \quad = \quad x \approx x' \wedge w \approx_F w' \tag{6.17}$$
$$tip(x) \circ w \approx_F (u' \circ v') \circ w' \quad = \quad tip(x) \circ w \approx_F u' \circ (v' \circ w') \tag{6.18}$$

At this stage, we have *three* inductive hypotheses in the H component of the procedure: (6.9), (6.10) and (6.11). The hypothesis (6.10) has been used in simplifying (6.15), and (6.11) in simplifying (6.17) and (6.18). The only remaining cases that need further work are (6.14) and (6.16). Program equations for them can be synthesized using the same process, but we get a clearer program if we (manually) postulate the lemmas:

$$nil \approx_L f(x) \quad = \quad \textbf{false} \tag{6.19}$$
$$f(x) \approx_L nil \quad = \quad \textbf{false} \tag{6.20}$$

These are proved in the standard fashion. Using them to simplify the equations results in the following final program:

$$\begin{aligned}
tip(x) \approx_F tip(x') &\rightarrow x \approx x' \\
tip(x) \approx_F tip(x') \circ w' &\rightarrow \textbf{false} \\
tip(x) \approx_F (u' \circ v') \circ w' &\rightarrow tip(x) \approx_F u' \circ (v' \circ w') \\
tip(x) \circ w \approx_F tip(x') &\rightarrow \textbf{false} \\
tip(x) \circ w \approx_F tip(x') \circ w' &\rightarrow x \approx x' \wedge w \approx_F w' \\
tip(x) \circ w \approx_F (u' \circ v') \circ w' &\rightarrow tip(x) \circ w \approx_F u' \circ (v' \circ w') \\
(u \circ v) \circ w \approx_F y &\rightarrow u \circ (v \circ w) \approx_F y
\end{aligned}$$

We also obtain the following inductive theorems as by products:

$$\begin{aligned}
f(x) \approx_L f(y) &\rightarrow x \approx_F y \\
x \cdot nil \approx_L f(y) &\rightarrow tip(x) \approx_F y \\
x \cdot f(w) \approx_L f(y) &\rightarrow tip(x) \circ w \approx_F y \\
nil \approx_L f(x) &\rightarrow \textbf{false} \\
f(x) \approx_L nil &\rightarrow \textbf{false}
\end{aligned}$$

This example is interesting in that we need to instantiate the variables x and y of the original specification in a controlled fashion to obtain a valid program. Note that we did not need to postulate an auxiliary function to calculate the fringe of a *list* of trees, as done in (Burstall and Darlington, 1977).

7. Generalization and Auxiliary Procedures

In this section, we describe some of the heuristics that can be applied to hypothesize program statements and inductive lemmas.

Suppose we wish to synthesize a program that doubles a natural number (in successor notation), without recourse to the addition function. Running the synthesis procedure with domain equations

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

and specification

$$x + x \quad = \quad d(x)$$

generates an infinite set of equations:

$$\begin{aligned} d(0) &= 0 \\ s(s(x)+x) &= d(s(x)) \\ d(s(0)) &= s(s(0)) \\ s(s(s(x)+x))) &= d(s(s(x))) \\ d(s(s(0))) &= s(s(s(s(0)))) \\ &\vdots \end{aligned}$$

There is, of course, little one can do with the resultant "program", which is no more than an *infinite* table lookup: $\{d(s^i(0)) = s^{2i}(0) \ : \ i \geq 0\}$. What is needed is some way of *guessing* the more general equation $d(s(x)) = s(s(d(x)))$.

We use two processes to generate hypotheses. The first involves generating critical pairs between equations; the second is a syntactic form of generalization, à la (Boyer and Moore, 1977; Arsac and Kodratoff, 1982). The intuition is that if we are dissatisfied from the computational point of view with the equations generated, we look for new equations between terms containing the defined function symbol in the hope of discovering a pattern. This approach was suggested in (Dershowitz and Pinchover, 1990).

For the first step, we overlap the smaller sides of the equations in the current partial program. For this purpose we use an ordering under which constructor terms are larger than terms containing the defined function applied to non-base cases: $d(0) \succ 0$, but $s(s(0)) \succ d(s(0))$, $s(s(s(0))))) \succ d(s(s(0)))$, etc. Using the equations in this direction brings patterns involving d to the fore. By overlapping the right-hand sides of $d(s(0)) = s(s(0))$ and $d(s(s(0))) = s(s(s(0))))$, we get a critical pair $d(s(s(0))) = s(s(d(s(0))))$. From $d(s(s(0))) = s(s(s(s(0))))$ and $d(s(s(s(0)))) = s(s(s(s(s(s(0))))))$, we get $d(s(s(s(0)))) = s(s(d(s(s(0))))))$, and so on.

For the second step, we generate *most specific generalizations* of pairs of equations, by replacing conflicting subterms with a new variable (see (Plotkin, 1970)). This process has been called "anti-unification"; given two terms s and t, it computes their greatest lower bound (*glb*) in the subsumption lattice. The above two critical pairs generate the hypothesis $d(s(x)) = s(s(d(x)))$. Applying $d(x) = x + x$, gives $s(x) + s(x) = s(s(x + x))$, which simplifies to $s(s(x) + x) = s(s(x + x))$, using the equation $x + s(y) = s(x + y)$, but no further (not knowing the inductive theorem $s(x) + y = s(x + y)$). Note that we are assuming $d(x) \succ x + x$ for the purposes of verification, which is the opposite direction of what was used for synthesis. Were this equation provable by deductive means, we would be finished; it is not, so the inductive proof method continues in the same manner, generating an infinite sequence of hypotheses:

$$\begin{aligned} s(s(x)+x) &= s(s(x+x)) \\ s(s(s(y))+y)) &= s(s(s(y)+y))) \\ &\vdots \end{aligned}$$

Clearly, we need to substitute the (missing) lemma $s(x) + y = s(x + y)$ for these instances. We employ the same generalization methods as for synthesis (see (Jantke, 1989; Lange, 1989)). An additional helpful technique is *cancellation*, as used in deduction, for example, in (Stickel, 1984). In particular, we can take advantage of constructors, replacing hypotheses of the from $c(s_1, \ldots, s_n) = c(t_1, \ldots, t_n)$ with n hypotheses $s_i = t_i$, when the constructor is free (Huet and Hullot, 1982). In the above case, we are free to strip off

matching outer s's from the generated hypotheses:

$$\begin{array}{rcl} s(x)+x & = & s(x+x) \\ s(s(y))+y & = & s(s(y)+y) \\ & \vdots & \end{array}$$

Generalizing, as before, leads to the hypothesis $s(x)+y = s(x+y)$, exactly what we were looking for.

With this added to the specification, the recursive program

$$\begin{array}{rcl} d(0) & = & 0 \\ d(s(x)) & = & s(s(d(x))) \end{array}$$

for d is finally proved correct. The first equation is a deductive consequence of the specification; the second is an inductive consequence.

Having succeeded in producing a program for doubling, a recursive program for halving can be generated from the *implicit* definition

$$\begin{array}{rcl} h(d(x)) & = & x \\ h(s(d(x))) & = & x \end{array}$$

The following sequence of equations is produced:

$$\begin{array}{rcl} h(0) & = & 0 \\ h(s(0)) & = & 0 \\ h(s(s(0))) & = & s(0) \\ h(s(s(s(0)))) & = & s(0) \\ h(s(s(s(s(0))))) & = & s(s(0)) \\ & \vdots & \end{array}$$

These equations suggest at least two hypotheses, namely:

$$\begin{array}{rcl} h(x) & = & h(s(x)) \\ s(h(x)) & = & h(s(s(x))) \end{array}$$

The former generalizes the equations

$$\begin{array}{rcl} h(0) & = & h(s(0)) \\ h(s(s(0))) & = & h(s(s(s(0)))) \end{array}$$

but is disproved, since (taking $x = s(0)$) it implies that $s(0) = 0$. The second hypothesis is obtained by looking at different pairs of equations (first and third, second and fourth, etc.) and generalizes the equations

$$\begin{array}{rcl} s(h(0)) & = & h(s(s(0))) \\ s(h(s(s(0)))) & = & h(s(s(s(s(0))))) \end{array}$$

It is proved immediately by induction, yielding the correct and complete program

$$\begin{array}{rcl} h(0) & = & 0 \\ h(s(0)) & = & 0 \\ h(s(s(x))) & = & s(h(x)) \end{array}$$

Most programs require auxiliary procedures, in addition to the specified top-level program. Two heuristics come into play here: The first is to abstract a subterm appearing

in a program, creating a subprogram to compute it (cf. (Kodratoff and Picard, 1983; Bellegarde, 1991)). The second is to compute two functions at once, or one function for two arguments, when expanding (unfolding) the definition of one leads to multiple applications of the same function (cf. (Burstall and Darlington, 1977; Feather, 1982; Reddy, 1989; Bellegarde, 1991)).

For example, suppose we have all three equations for addition, and wish to manufacture a program $q(x)$ for squaring from the following equations for multiplication:

$$\begin{array}{rcl} x \times 0 & = & 0 \\ x \times s(y) & = & (x \times y) + x \\ s(x) \times y & = & (x \times y) + y \\ x \times x & = & q(x) \end{array}$$

The synthesis procedure with precedence $\times > + > q$ will generate the following facts (among others):

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

Noting the repeating left-hand side subterm pattern $(x+z)+z$ suggests the introduction of an ancillary function:

$$(x + z) + z \quad = \quad p(x, z)$$

Synthesizing p in the same manner as we synthesized d, gives

$$\begin{array}{rcl} p(x, 0) & = & x \\ p(x, s(y)) & = & s(s(p(x, y))) \end{array}$$

Letting p be a smaller operator symbol than q (since it is all right for q to be defined in terms of p), we get

$$q(s(x)) \quad = \quad s(p(q(x), x))$$

With this equation, used from left to right, equations like $s(s(s((q(s(y)) + y) + y))) = q(s(s(y)))$ simplify away. Together, the equations for p and q constitute a program for squaring.

Alternatively, suppose we know that + is associative:

$$(x + y) + z \quad = \quad x + (y + z)$$

with the left side greater than the right. Then the consequences

$$\begin{array}{rcl} s(q(x) + (x + x)) & = & q(s(x)) \\ s(s(s(q(s(y)) + (y + y)))) & = & q(s(s(y))) \end{array}$$

suggest the auxiliary function:

$$x + x \quad = \quad d(x)$$

That leaves us with the following squaring program:

$$\begin{array}{rcl} q(0) & = & 0 \\ q(s(x)) & = & s(q(x) + d(x)) \end{array}$$

8. Discussion

Rewriting is a powerful tool in equational reasoning, in which orderings on terms play a central role. In ordered rewriting, orderings are used to determine the direction of computation, by providing a suitable concept of what makes one term "simpler" than another. Ordered rewriting is more flexible than standard rewriting, since it allows the same equation to be used sometimes in one direction, and sometimes in the other. In theorem proving, as well, orderings are crucial for incorporating powerful simplification rules in complete inference systems. Last, but not least, orderings supply us with a basis for inductive proofs, which are essential for proving properties of programs.

The approach to synthesis described here comprises both formal and informal aspects. We use equational reasoning and mathematical induction to guarantee correctness of the synthesized programs. On the other hand, we apply heuristics to *suggest* equations for incorporation in developing programs, as well as for forming lemmas needed in inductive proofs.

We have only considered rewriting with equations. Conditional rewriting and goal solving may provide a better combination of functional and logic programming than purely equational programs; see, for instance, (Dershowitz and Plaisted, 1988). Conditional synthesis, however, would necessitate more powerful deductive and inductive methods for handling conditional equations, such as have been investigated in (Kounalis and Rusinowitch, 1991; Ganzinger, 1991; Bronsard and Reddy, 1992). More elaborate generalization methods would also be required.

An interactive program transformation system called "Focus" has been implemented at the University of Illinois based on the techniques presented here. The system incorporates "oriented" rewriting techniques (a special case of the ordered rewriting techniques considered here) and also several extensions for conditional and first-order reasoning. It has been used to synthesize several interesting examples including some reasonably large programs (Reddy, 1988; Reddy, 1990a; Reddy, 1991).

References

ACM, (1991). *Symp. Partial Evaluation and Semantics-Based Program Manipulation.* SIGPLAN Notices, 26(9):1991.

Arsac, J., Kodratoff, Y., (1982). Some techniques for recursion removal from recursive functions. *ACM Trans. Program. Lang. Systems*, **4**(2), 295–322.

Bachmair, L., (1988). Proof by consistency. In *Symp. on Logic in Comp. Science.* IEEE.

Bachmair, L., Dershowitz, N., (to appear). Equational inference, canonical proofs, and proof orderings. *J. ACM*, .

Bachmair, L., Dershowitz, N., Plaisted, D. A., (1989). Completion without failure. In Aït-Kaci, H., Nivat, M., eds., *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, chapter 1, pages 1–30. Academic Press.

Bellegarde, F., (1991). Program transformation and rewriting. In Book, R., ed., *Fourth Intern. Conf. on Rewriting Techniques and Applications*, volume 488 of *Lect. Notes in Comp. Science*, pages 226–239. Springer-Verlag.

Bergstra, J. A., Klop, J. W., (1986). Conditional rewrite rules: Confluency and termination. *J. of Computer and System Sciences*, **32**, 323–362.

Bibel, W., Hörnig, K. M., (1984). LOPS - A system based on a strategical approach to program synthesis. In Biermann, A. W., Guiho, G., Kodratoff, Y., eds., *Automatic Program Construction Techniques*, chapter 3, pages 69–90. New York: MacMillan Pub. Co.

Bird, R., Wadler, P., (1988). *Introduction to Functional Programming.* London: Prentice-Hall International.

Bjorner, D., Erschov, A. P., Jones (eds), N. D., (1988). *Partial Evaluation and Mixed Computation.* North-Holland.

Boyer, R. S., Moore, J. S., (1977). A lemma driven automatic theorem prover for recursive function theory. In *Intern. Joint Conf. on Artificial Intelligence*, pages 511–519, Cambridge, MA.

Brand, D., (1975). Proving theorems with the modification method. *SIAM J. on Computing*, **4**, 412–430.

Bronsard, F., Reddy, U. S., (1991). Conditional rewriting in Focus. In Kaplan, S., Okada, M., eds., *Conditional and Typed Rewriting Systems — Second International CTRS Workshop*, volume 516 of *Lect. Notes in Comp. Science*, pages 2–13. Springer-Verlag.

Bronsard, F., Reddy, U. S., (1992). Reduction techniques for first-order reasoning. In Rusinowitch, M., Rémy, J. L., eds., *Conditional Term Rewriting Systems*, volume 656 of *Lect. Notes in Comp. Science*, pages 242–256. Springer-Verlag.

Bündgen, R., Küchlin, W., (1989). Computing ground reducibility and inductively complete positions. In Dershowitz, N., ed., *Rewriting Techniques and Applications*, volume 355 of *Lect. Notes in Comp. Science*, pages 59–75. Springer-Verlag.

Burstall, R. M., Darlington, J., (1977). A transformation system for developing recursive programs. *J. ACM*, **24**(1), 44–67.

Comon, H., (1990). Solving inequations in term algebras (Preliminary version). In *Fifth Ann. Symp. on Logic in Comp. Science*, pages 62–69, Philadelphia, PA. IEEE.

Darlington, J., (1981). The structured description of algorithm derivations. In de Bakker, J. W., van Vliet, J. C., eds., *Algorithmic Languages*, pages 221–250. North-Holland.

Dershowitz, N., (1982). Applications of the Knuth–Bendix completion procedure. In *Proc. of the Seminaire d'Informatique Theorique, Paris*, pages 95–111. (Also vailable as Technical Report ATR-83(8478)-2, Information Sciences Research Office, The Aerospace Corporation, El Segundo, CA.)

Dershowitz, N., (1985). Computing with rewrite systems. *Information and Control*, **65**(2/3), 122–157.

Dershowitz, N., (1985). Synthesis by completion. In *Proc. Ninth Intern. Joint Conf. on Artificial Intelligence*, pages 208–214.

Dershowitz, N., (1987). Termination of rewriting. *J. Symbolic Computation*, **3**, 69–116.

Dershowitz, N., (1989). Completion and its applications. In *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, pages 31–86. San Diego: Academic Press.

Dershowitz, N., Jouannaud, J.-P., (1990). Rewrite systems. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. Amsterdam: North-Holland.

Dershowitz, N., Pinchover, E., (1990). Inductive synthesis of equational programs. In *Eighth National Conf. on Artificial Intelligence*, pages 234–239, Boston, MA. AAAI.

Dershowitz, N., Plaisted, D. A., (1988). Equational programming. In Hayes, J. E., Michie, D., Richards, J., eds., *Machine Intelligence 11: The logic and acquisition of knowledge*, chapter 2, pages 21–56. Oxford: Oxford Press. To be reprinted in *Logical Foundations of Machine Intelligence*, Horwood.

Deville, Y., (1990). *Logic Programming: Systematic Program Development.* Wokingham: Addison-Wesley.

Feather, M. S., (1982). A system for assisting program transformation. *ACM Trans. Program. Lang. Systems*, **4**(1), 1–20.

Fribourg, L., (1989). A strong restriction of the inductive completion procedure. *J. Symbolic Computation*, **8**(3), 253–276.

Fronhöfer, B., Furbach, U., (1986). Knuth-Bendix completion versus fold/unfold: A comparative study in program synthesis. In Rollinger, C., Horn, W., eds., *Proc. of the Tenth German Workshop on Artificial Intelligence*, pages 289–300.

Ganzinger, H., (1991). A completion procedure for conditional equations. *J. Symbolic Computation*, **11**, 51–81.

Goldammer, U., (1992). A method for the inductive synthesis of rewrite programs based on Knuth-Bendix completion techniques. GOSLER Report 06/92, Technische Hochschule Leipzig, Leipzig, Germany.

Gramlich, B., (1989). Induction theorem proving using refined unfailing completion techniques. Technical Report SR89-14, Universität Kaiserslautern, Germany.

Hoare, C. A. R., (1971). Procedures and parameters: An axiomatic approach. In Engeler, E., ed., *Symp. Semantics of Algorithmic Languages*, volume 188 of *Lect. Notes in Math.*, pages 102–116. Springer-Verlag.

Hogger, C. J., (1976). Derivation of logic programs. *J. ACM*, **28**(2), 372–392.

Hsiang, J., Dershowitz, N., (1983). Rewrite methods for clausal and non-clausal theorem proving. In *10th Intern. Colloq. Automata, Languages and Programming*, volume 154 of *Lect. Notes in Comp. Science*, pages 331–346. Springer-Verlag.

Hsiang, J., Rusinowitch, M., (1987). On word problems in equational theories. In Ottmann, T., ed., *14th Intern. Colloq. Automata, Languages and Programming*, volume 267 of *Lect. Notes in Comp. Science*, pages 54–71. Springer-Verlag.

Hsiang, J., Rusinowitch, M., (1991). A new method for establishing refutational completeness in theorem proving. *J. ACM*, **38**(3), 559–587.

Huet, G., Hullot, J.-M., (1982). Proofs by induction in equational theories with constructors. *J. Comp. and System Sciences*, **25**, 239–266.

Huet, G., Oppen, D. C., (1980). Equations and rewrite rules: A survey. In Book, R., ed., *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. New York: Academic Press.

Jantke, K. P., (1989). Algorithmic learning from incomplete information: Principles and problems. In Dassow, J., Kelemen, J., eds., *Machines, Languages, and Complexity (Selected Contributions of the 5th International Meeting of Young Computer Scientists, Smolenice, Czechoslovakia, November 1988)*, volume 381 of *Lect. Notes in Comp. Science*, pages 188–207. Springer-Verlag.

Jouannaud, J.-P., Kounalis, E., (1989). Automatic proofs by induction in equational theories without constructors. *Information and Computation*, **82**, 1–33.

Kamin, S., Lévy, J.-J., (1980). Two generalizations of the recursive path ordering. Unpublished note, Department of Computer Science, University of Illinois, Urbana, IL.

Kaplan, S., (1987). Simplifying conditional term rewriting systems: Unification, termination and confluence. *J. Symbolic Computation*, **4**, 295–334.

Kapur, D., Musser, D. R., (1987). Proof by consistency. *Artificial Intelligence*, **31**(2), 125–157.

Kapur, D., Srivas, M., (1985). A rewrite rule based approach for synthesizing data types. In *Intern. Joint Conf. Theory and Practice of Softw. Development (TAPSOFT)*, volume 186 of *Lect. Notes in Comp. Science*, pages 188–207. Springer-Verlag.

Kapur, D., Narendran, P., Otto, F., (1987). On ground confluence of term rewriting systems. Technical Report 87-6, General Electric R & D Center, Schenectady, New York. To appear in *Information and Computation*.

Kapur, D., Narendran, P., Zhang, H., (1991). Automating inductionless induction using test sets. *J. Symbolic Computation*, **11**, 83–112.

Klop, J. W., (1992). Term rewriting systems. In Abramsky, S., Gabbay, D. M., Maibaum, T. S. E., eds., *Handbook of Logic in Computer Science*, volume 1, chapter 6. Oxford: Oxford University Press.

Knuth, D., Bendix, P., (1970). Simple word problems in universal algebras. In Leech, J., ed., *Computational Problems in Abstract Algebra*, pages 263–297. Oxford: Pergamon Press.

Kodratoff, Y., Picard, M., (1983). Complétion de systèmes de réécriture et synthèse de programmes à partir deleurs spécifications. *Bigre*, **35**.

Kounalis, E., Rusinowitch, M., (1987). On word problems in Horn theories. In Kaplan, S., Jouannaud, J.-P., eds., *Conditional Term Rewriting Systems*, volume 308 of *Lect. Notes in Comp. Science*, pages 144–160. Springer-Verlag.

Kounalis, E., Rusinowitch, M., (1991). Inductive reasoning in conditional theories. In Kaplan, S., Okada, M., eds., *Conditional and Typed Rewriting Systems — Second International CTRS Workshop*, volume 516 of *Lect. Notes in Comp. Science*. Springer-Verlag.

Kounalis, E., Zhang, H., (1985). A general completeness test for equational specifications. In *Hungarian Conference of Computer Science*. (Also available as Tech. Report CRIN [85-R-05], University of Nancy, Nancy, France.)

Küchlin, W., (1989). Inductive completion by ground proof transformation. In Aït-Kaci, H., Nivat, M., eds., *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, pages 211–245. San Diego: Academic Press.

Lange, S., (1989). Towards a set of inference rules for solving divergence in Knuth-Bendix completion. In Jantke, K. P., ed., *Proceedings of the International Workshop on Analogical and Inductive Inference*, volume 397 of *Lect. Notes in Comp. Science*, pages 304–316. Springer-Verlag.

Manna, Z., (1974). *Mathematical Theory of Computation*. New York: McGraw-Hill.

Manna, Z., Waldinger, R., (1980). A deductive approach to program synthesis. *ACM Trans. Program. Lang. Systems*, **2**(1), 90–121.

Martin, U., Nipkow, T., (1990). Ordered completion. In Stickel, M., ed., *Conf. on Automated Deduction*, Lect. Notes in Comp. Science, pages 366–380.

Musser, D. R., (1980). On proving inductive properties of abstract data types. In *ACM Symp. on Princ. of Program. Lang.*, pages 154–162. ACM.

Nieuwenhuis, R., Orejas, F., (1991). Clausal rewriting. In Kaplan, S., Okada, M., eds., *Conditional and Typed Rewriting Systems — Second International CTRS Workshop*, volume 516 of *Lect. Notes in Comp. Science*, pages 246–258. Springer-Verlag.

Paulson, L. C., (1991). *ML for the Working Programmer*. Cambridge: Cambridge Univ. Press.

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

Peterson, G. E., (1990). Complete sets of reductions with constraints. In Stickel, M., ed., *10th Intern. Conf. on Automated Deduction*, Lect. Notes in Comp. Science, pages 381–395.

Plaisted, D., (1985). Semantic confluence tests and completion methods. *Information and Control*, **65**, 182–215.

Plotkin, G., (1970). Lattice theoretic properties of subsumption. Technical Report MIP-R-77, University of Edinburgh, Edinburgh, Scotland.

Reddy, U. S., (1988). Transformational derivation of programs using the Focus system. *SIGSOFT Software Engineering Notes*, **13**(5), 163–172. (Proceedings, ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. on Practical Software Development Environments, also published as SIGPLAN Notices, Feb. 1989.)

Reddy, U. S., (1989). Rewriting techniques for program synthesis. In Dershowitz, N., ed., *Rewriting Techniques and Applications*, volume 355 of *Lect. Notes in Comp. Science*, pages 388–403. Springer-Verlag.

Reddy, U. S., (1990). Formal methods in transformational derivation of programs. *Software Engineering Notices*, **15**(4), 104–114. (Proceedings of the ACM SIGSOFT Workshop on Formal Methods in Software Development.)

Reddy, U. S., (1990). Term rewriting induction. In Stickel, M., ed., *10th Intern. Conf. on Automated Deduction*, volume 449 of *Lect. Notes in Artificial Intelligence*, pages 162–177. Springer-Verlag.

Reddy, U. S., (1991). Design principles for an interactive program derivation system. In Lowry, M., McCartney, R. D., eds., *Automating Software Design*, chapter 18. AAAI Press.

Robinson, G., Wos, L., (1969). Paramodulation and theorem-proving in first order theories with equality. In Meltzer, B., Michie, D., eds., *Machine Intelligence 4*, pages 135–150. Edinburgh, Scotland: Edinburgh University Press.

Scott, D., (1976). Data types as lattices. *SIAM J. Computing*, **5**(3), 522–587.

Smith, D., (1985). Top-down synthesis of divide and conquer algorithms. *Artificial Intelligence*, **27**, 43–96.

Stickel, M. E., (1984). A case study of theorem proving by the Knuth Bendix method discovering that $x^3 = x$ implies ring commutativity. In Shostak, R. E., ed., *Proceedings of the Seventh International Conference on Automated Deduction*, volume 170 of *Lect. Notes in Comp. Science*, pages 248–259. Springer-Verlag.

Tamaki, H., Sato, T., (1984). Unfold/fold transformation of logic programs. In *Intern. Conf. on Logic Programming*, pages 127–138.

Zhang, H., Kapur, D., (1988). First-order theorem proving using conditional rewrite rules. In Lusk, E., Overbeek, R., eds., *9th Intern. Conf. on Automated Deduction*, pages 1–20. Springer-Verlag.