Deductive and Inductive Synthesis of Equational Programs
—Draft—

Nachum Dershowitz and **Uday S. Reddy**
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
Net: {nachum,reddy}@cs.uiuc.edu

August 3, 1992

Abstract

An equational approach to the synthesis of functional and logic program is taken. Typically, the synthesis task involves finding equations which make the given specification an inductive theorem. To synthesize such programs, induction is necessary. We formulate efficient procedures for inductive proof as well as 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 not adequate to come with a program.

1 Introduction

In a seminal piece of work, Burstall and Darlington [BD77] 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 forwards ("unfolding") and backwards ("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. (Cf. [BW88]). It has also been adapted to reasoning about logic programs, see, for example, [Dev90, Hog76, TS84]. Significant effort has been devoted to building automated systems based on the methodology [Dar81, Fea79, Fea82]. Partial evaluation systems, which have been increasingly successful in recent times [ACM91, BEJ88], are also based on the unfold-fold method. In building reliable general-purpose program synthesis systems, however, several issues arise:

- How to determine if the transformed programs are correct? (While the soundness is immediate from the technique, termination and completeness are still concerns.)
- How to control the application of equations? (Naive application of equations leads to large search spaces. The controlled application used by Burstall-Darlington does not handle all problems.)

- How does the method generalize to forms of programs (and logics) other than equational ones? (For instance, 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 techniques of deductive synthesis, like [BH84, MW80, Smi90]?

In attempting to answer such questions, we are led to the framework of *term rewriting*, which is the best known technique of controlled equational reasoning. Term rewriting was first used in automated reasoning by Knuth and Bendix [KB70] for solving word problems in equational theories. Two fundamental operations underlie the technique: *rewriting* and *superposition*. Rewriting uses a terminating rewrite system to rewrite a term to a normal form. Superposition uses two given rewrite rules to deduce a new rewrite rule. The combination of the two techniques achieves an extremely high performance in equational reasoning. In recent work, term rewriting techniques have been extended to deal with unoriented equations [BDP89, HR87, MN90], conditional equations [BK86, Gan91, Kap87, KR87], and first-order reasoning [BR91, HD83, HR86, NO91, ZK88]. See [Der89, DJ90, HO80] for accessible surveys of this rapidly developing area.

The contributions of this paper are three fold: 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 [Der82a, DP90, Red89, Red90b]. In the cited work, we treat rewrite systems, and, here, generalize the techniques to unoriented equations using the notion of "ordered rewriting". The application of ordered rewriting to program synthesis and inductive proofs has also been considered in [Bac88, Bel91a, Gra89]. [FF86] compares rewriting techniques with the plain equational methods of Burstall and Darlington.

2 Overview of program synthesis

Assume that we wish to synthesize a program for some function f and are given a specification S of the function f together with an axiomatization E of the problem domain. There are two ways to think about the program synthesis process. First, we can try to generate all interesting logical consequences of $S \cup E$ in the hope of eventually obtaining some set of equations which serves as a program for f. Second, we can try to reduce the specification S into simpler equations, using E, in the hope of eventually obtaining equations simple enough to serve as a program for f. Apparently, the second view is more goal-directed than the first. However, in the context of equational reasoning, both the views produce similar results modulo issues of search space.

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

$$append(X, nil) = X \quad (1)$$

$$append(nil, Y) = Y \quad (2)$$

$$append(A \cdot U, Y) = A \cdot append(U, Y) \quad (3)$$

$$append(append(X, Y), Z) = append(X, append(Y, Z)) \quad (4)$$

$$reverse(nil) = nil \quad (5)$$
$$reverse(A \cdot U) = append(reverse(U), A \cdot nil) \quad (6)$$

We would like to synthesize a more efficient program for *reverse* using a new function *rev*, specified by:

$$rev(X, Y) = append(reverse(X), Y) \quad (7)$$

By noting that the subterm $reverse(X)$ can be simplified using the defining equations of *reverse*, if X is instantiated to *nil* and $A \cdot U$ respectively, we obtain:

$$\begin{aligned}
rev(nil, Y) &= append(reverse(nil), Y) \\
&= append(nil, Y) \\
&= Y \\
rev(A \cdot U, Y) &= append(reverse(A \cdot U), Y) \\
&= append(append(reverse(U), A \cdot nil), Y) \\
&= append(reverse(U), append(A \cdot nil, Y)) \\
&= append(reverse(U), A \cdot Y) \\
&= rev(U, A \cdot Y)
\end{aligned}$$

All the steps use axioms to replace "equals by equals", except for the last step which uses the original specification for a smaller instance. (Such use of the original specification is termed "folding" in [BD77].) Note also that all these steps can be viewed as either forward or backward reasoning steps. In the forward reasoning view, we depend on an appropriate synthesis procedure to generate only "interesting" consequences of the specification. We also need some other mechanism to verify whether a correct (terminating and complete) program is obtained. In the backward reasoning view, we think of these same steps as reducing the specification to simpler equations. The last folding step needs to be justified more carefully in this view. Because our task is to produce program equations which verify the specification, we cannot use the specification arbitrarily in the synthesis process. The justification is that, in producing the program for the instance $X = A \cdot U$, we can assume the specification as an *inductive hypothesis* for the smaller instance $X = U$. Thus, folding steps correspond to induction steps in the backward reasoning view. For the price of carrying out this additional inductive reasoning, we obtain an important benefit: the synthesized program is guaranteed to be correct.

Yet another useful equational consequence is derivable from the specification. Notice that the right hand side of the specification is simplifiable by the first axiom if Y is instantiated to *nil*:

$$\begin{aligned}
rev(X, nil) &= append(reverse(X), nil) \\
&= reverse(X)
\end{aligned}$$

Oriented backwards, as $reverse(X) = rev(X, Nil)$, this equation gives us a new program for *reverse* in terms of the synthesized function *rev*. This is a forward reasoning step which has no justification in the backward reasoning view. Our program synthesis procedure mixes forward and backward reasoning steps—as well as heuristic steps—to achieve a viable and automatic procedure.

Many program synthesis tasks involve conditional reasoning in addition to equational reasoning. Even though term rewriting techniques have been extended to conditional equations as well as first-order clauses, we do not wish to get into these technical areas in this paper. Instead, we will use an equational axiomatization of boolean algebras and the trick that $U \supset V$ is equivalent to

$$
\begin{array}{rcl}
U \wedge \mathbf{true} & = & U \\
U \wedge \mathbf{false} & = & \mathbf{false} \\
U \wedge U & = & U \\
U \Leftrightarrow \mathbf{true} & = & U \\
U \Leftrightarrow U & = & \mathbf{true} \\
(U \Leftrightarrow V) \wedge W & = & (U \wedge W) \Leftrightarrow (V \wedge W) \Leftrightarrow W \\
\neg U & = & U \Leftrightarrow \mathbf{false} \\
U \vee V & = & (U \wedge V) \Leftrightarrow U \Leftrightarrow V \\
U \supset V & = & (U \wedge V) \Leftrightarrow U \\
U \wedge V & = & V \wedge U \\
(U \wedge V) \wedge W & = & U \wedge (V \wedge W) \\
U \Leftrightarrow V & = & V \Leftrightarrow U \\
(U \Leftrightarrow V) \Leftrightarrow W & = & U \Leftrightarrow (V \Leftrightarrow W) \\
U \vee V & = & V \vee U \\
(U \vee V) \vee W & = & U \vee (V \vee W)
\end{array}
$$

Figure 1: Equational axiomatization of propositional calculus

$(U \wedge V) = V$ where $=$ denotes equality, in this case, 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 $\Leftrightarrow$. Figure 1 gives an equational axiomatization of propositional calculus in this notation. (Cf. [HD83].)

Consider the following axiomatization of addition, multiplication and comparison of natural numbers in the unary number system, wherein the number n is represented as a sum of n 1's:

$$
\begin{array}{rcll}
M + 0 & = & M & (8) \\
M + N & = & N + M & (9) \\
(M + N) + K & = & M + (N + K) & (10) \\
M \times 0 & = & 0 & (11) \\
M \times (N + 1) & = & M \times N + M & (12) \\
M \approx M & = & \mathbf{true} & (13) \\
M + K \approx N + K & = & M \approx N & (14) \\
M \leq M + N & = & \mathbf{true} & (15)
\end{array}
$$

We want to synthesize a program for natural number division specified by

$$
div(X, Y + 1, Q, R) \quad = \quad R \leq Y \wedge X \approx (Y + 1) \times Q + R \qquad (16)
$$

We proceed as in the *rev* example, and instantiate Q to 0 and $N + 1$ to use the first two defining

axioms of $\times$ respectively:

$$\begin{aligned} div(X, Y+1, 0, R) &= R \leq Y \wedge X \approx (Y+1) \times 0 + R \\ &= R \leq Y \wedge X \approx 0 + R \\ &= R \leq Y \wedge X \approx R + 0 \\ &= R \leq Y \wedge X \approx R \\ div(X, Y+1, N+1, R) &= R \leq Y \wedge X \approx (Y+1) \times (N+1) + R \\ &= R \leq Y \wedge X \approx (Y+1) \times N + (Y+1) + R \\ &= R \leq Y \wedge X \approx ((Y+1) \times N + R) + (Y+1) \end{aligned}$$

For the first case, we can instantiate R to X for using the domain fact (13) and Y to $X + Z$ for using the axiom (15). This gives the more compact version of the equation:

$$div(X, X + Z + 1, 0, X) = \mathbf{true}$$

For the second case, we can instantiate X to $U+Y+1$ for using the axiom (14) with the substitution $[M \mapsto U,\ K \mapsto Y+1,\ N \mapsto (Y+1) \times N + R]$. This gives:

$$\begin{aligned} div(U + Y + 1, Y + 1, N + 1, R) &\rightarrow R \leq Y \wedge U \approx (Y+1) \times N + R \\ &\rightarrow div(U, Y + 1, N, R) \end{aligned}$$

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

$$\begin{aligned} div(X, X + Z + 1, 0, X) &\rightarrow \mathbf{true} \\ div(U + Y + 1, Y + 1, N + 1, R) &\rightarrow div(U, Y + 1, N, R) \end{aligned}$$

can be viewed as a logic program for division. (See [Der85] for a discussion of how logic programs are treated in the equational framework.)

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. The set of terms over Σ (respecting arities) is denoted T_Σ (called the set of *ground* terms). The set of (free) terms over Σ using variables from a set X is denoted $T_\Sigma(X)$. We assume a countable set of variables X and refer to members of $T_\Sigma(X)$ as simply *terms*, or *free terms*, for clarification.

An *equation* is a pair of terms written as $r = s$. Given a set of equations E and terms t and u, $E \vdash t = u$ if and only if there are terms t_0, t_1, ..., t_n $(n \geq 0)$ such that

$$t \equiv t_0 =_E t_1 =_E \ldots =_E t_n \equiv u$$

where $=_E$ is the "replacing equals by equals" relation of E. A sequence such as the one exhibited is called an "equational proof". The standard relational notations $=_E^+$ and $=_E^*$ are used to denote the transitive and reflexive-transitive closure of $=_E$ respectively. Thus, $E \vdash t = u$ iff $t =_E^* u$.

To treat equations as programs, we need the notion of a *rewrite relation*. This is defined in terms of a well-founded order with certain extra properties stated below. Let $\succ$ be such an order. We say that t *rewrites* to u and write $t \rightarrow_E u$ if $t =_E u$ and $t \succ u$. 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 $t_0 \rightarrow_E t_1 \rightarrow \ldots$ is finite and results in a normal form (which may not be unique). Thus, our equational programs are always terminating by definition. In equation $t = u$ is said to have a *rewrite proof* if there are terms t_0, t_1, ..., t_n and u_0, u_1, ... u_m such that

$$t \equiv t_0 \rightarrow_E t_1 \rightarrow_E \ldots \rightarrow_E t_n \equiv u_m \leftarrow_E \ldots \leftarrow_E u_1 \leftarrow_E u_0 \equiv u$$

where $\leftarrow_E$ is the relational inverse of $\rightarrow_E$. Thus, a rewrite proof is an equational proof that rewrites both t and u to some common normal form. As usual, $\rightarrow_E^+$ and $\rightarrow_E^*$ are used to denote the transitive and reflexive-transitive closures of $\rightarrow_E$ respectively. In addition, $\leftrightarrow_E$ denotes the symmetric closure of $\rightarrow_E$, *i.e.*, $t \leftrightarrow_E u$ iff $t =_E u$ and either $t \succ u$ or $u \succ t$.

If $r = s$ is an equation in E such that $r \succ s$, we also write it 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. But, this may be to rewrite an instance of r to s in some cases, and to rewrite an instance of s to r in others. For example, consider the commutativity equation $x + y = y + x$. Since the equation is symmetric, orienting it in any direction results in infinite rewrite sequences. On the other hand, specific instances of the equation, $t + u = u + t$, may achieve a reduction in one direction or the other. For instance, using a lexicographic order for $\succ$, $t + u \succ u + t$ iff $t \succ u$. In particular, we ensure that the order $\succ$ is total on ground terms. So, all ground instances of equations are obtainable by rewriting.

The conventional term rewriting theory deals with *rewrite systems*, *i.e.*, sets of equations $r \rightarrow s$ which are 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 proposed in [BDP89, HR87]. This form of rewriting is often called *ordered rewriting*. The results of this paper generalize our previous results [Der82a, Der85, DP90, Red89, Red90b] to the framework of ordered rewriting.

The mixing of programs and program synthesis with termination issues needs some explanation. Requiring 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 which do not terminate but make progress indefinitely. Programs in lazy functional languages [BW88] often exhibit this property. We envisage that the techniques of this paper will eventually be extended to such programs by suitable relaxation of the termination requirements. (Cf. [DK89].) 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 in essential role in the problem specification for program synthesis as well as in the synthesis process itself.

3.1 Orderings

We now state the required properties of the well-founded order $\succ$. A well-founded order $>$ on ground terms is called a *complete simplification ordering* if

- it is total on ground terms,

- it has the replacement property ($s > s'$ implies $t[s] > t[s']$), and
- it has the subterm property ($t > s$ whenever s is a proper subterm of t).

The order $\succ$ is defined on terms by $t \succ u$ iff $t\sigma > u\sigma$ for all ground substitutions σ, where $>$ is a complete simplification ordering. Note that $\succ$ also inherits the replacement and subterm properties. In addition, it also has the substitution property: $t \succ u$ implies $t\theta \succ u\theta$ for all substitutions θ. (In practice, it suffices to use some simplification ordering on free terms for $\succ$, provided it has the substitution property and $t \succ u$ implies $t\sigma > u\sigma$ for all ground substitutions σ.)

Of particular interest in program synthesis is the lexicographic *path ordering* [Der82b, KL80]. Assume a total order on function symbols referred to as a "precedence order". Then, the lexicographic path order $\succ$ is defined inductively by $t \equiv f(t_1, \ldots, t_m) \succ g(u_1, \ldots, u_n) \equiv u$ iff $t \succ u_i$ for all i and

- $t_i \succeq u$,
- $f > g$ in the precedence order, or
- $f = g$, $m = n$ and $\langle t_1, \ldots, t_m \rangle$ is greater than $\langle u_1, \ldots, u_m \rangle$ by the lexicographic extension of $\succ$.

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). Note that the lexicographic path order satisfies all the requirements above. It has the replacement, subterm and substitution properties and it is total on ground terms. Thus, the lexicographic path order serves as both the $\succ$ order on free terms and $>$ order on ground terms.

We illustrate the path order with examples. Consider the equations (1–6) and the precedence order

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

With the corresponding lexicographic path order, every left hand side is greater than the corresponding right hand side. For example, considering (1),

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

because X is a subterm of $append(X, nil)$. For the equation (4),

$$append(append(X, Y), Z) \succ append(X, append(Y, Z))$$

because $\langle append(X, Y), Z \rangle$ is greater than $\langle X, append(Y, Z) \rangle$. ($append(X, Y) \succ X$ by the subterm property.) For the equation (6),

$$reverse(A \cdot U) \succ append(reverse(U), A \cdot nil)$$

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

To handle the specification (7) of rev, we must extend the precedence order to include rev. A good heuristic in choosing precedences is that a symbol f may be greater than all the symbols

which may be introduced during the evaluation of $f(t_1, \ldots, t_n)$. Since the evaluation of $rev(t, u)$ must not introduce *reverse* and *append*, but may introduce $\cdot$ and nil, we choose the order

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

Note that

$$append(reverse(X), Y) \succ rev(X, Y)$$

by this extension. Thus, the specification (7) cannot be used left-to-right in evaluating terms of the form $rev(t, u)$. This defines the problem for the program synthesis procedure. It must find simpler equations which can be used to evaluate $rev(t, u)$.

3.2 Programs

An equational theory E is said to be *confluent* with respect to $\succ$ if, whenever $t =^*_E u$, there is a rewrite proof of $t = u$. It is said to be *ground confluent* if this property holds for all ground terms. Note that confluence implies that all terms have unique normal forms. Similarly, ground confluence implies that all ground terms have unique normal forms.

Definition 1 *An* equational program *is an equational theory E together with a complete simplification ordering $>$ such that E is ground confluent.*

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

Ground confluence is not a decidable property. On the other hand, confluence is decidable (for decidable orderings) and it 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 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. (Cf. Section 5.) The ground confluence of axioms can then be ensured by checking confluence. For example, considering the equational theory (1–6), the equations (1) and (3) define *append*, and the equations (5) and (6) define *reverse*. The rest are inductive theorems. Since no two left hand sides (greater sides) of the axioms have a common instance, they form a confluent theory and, so, the whole theory 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 $t \in \Phi$ 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 model equivalences over constructor terms in which case only a subset of constructor terms may be included in Ψ. For example, considering the axioms (8–10) 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 total functions. For example, the natural number axioms (13–15) only model the true cases of comparisons. It is not, in general, possible to specify the sets Φ and Ψ in a mechanically verifiable fashion, but [Der85] gives methods for some important cases.

4 Superposition and case-based reasoning

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 the instantiations which 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 which make further simplifications possible. For example, in the synthesis of *rev*, we chose instantiations which enable simplification by the axioms (1) and (3). In this section, we define formal methods for such instantiation.

The conventional inference rule for equational reasoning is *paramodulation* [RW69]:

$$PM \qquad \frac{l = r \quad t[s] = u}{t[r]\theta = u\theta} \qquad \text{if } \theta = \text{mgu}(l, s)$$

where *mgu* denotes the most general unifier. Thus, unification of the subterm s with l suggests how the variables in s should be instantiated so as to apply the equation $l = r$. However, unrestricted use of paramodulation generates too many consequences not all of which are useful. For instance, the right hand side of the specification (7) can be unified with the right hand sides (or some subterms) of all the axioms (1–4) and (6).

Since our equational theories are ground confluent with respect to the complete reduction order $>$, we can restrict attention to only certain uses of paramodulation. Suppose σ is a ground substitution that is more specific than θ. Then the effect of a paramodulation inference is to simplify the two step proof $t[r]\sigma =_E t[l]\sigma =_E u\sigma$ to a single step $t[r]\sigma =_E u\sigma$. (Note that $l\sigma \equiv s\sigma$ by virtue of σ being more specific than the mgu θ.) But, if $t[r]\sigma > t[l]\sigma$ or $u\sigma > t[l]\sigma$, then the initial proof is already a rewrite proof and nothing is achieved by the new equation. Thus, we only need to use the paramodulation inference if there is a ground substitution σ such that $t[l]\sigma > t[r]\sigma$ and $t[s]\sigma > u\sigma$. In terms of the order $\succ$, this means $t[l]\theta \not\succ t[r]\theta$ and $t[s]\theta \not\succ u\theta$. In other words $t[l]\theta$ is *maximal* in the equation $t[l]\theta = t[r]\theta$ and $t[s]\theta$ is maximal in $t[s]\theta = u\theta$. The restriction of paramodulation to this condition is called *ordered superposition*:

$$Sup \qquad \frac{l = r \quad t[s] = u}{t[r]\theta = u\theta} \qquad \text{if } s \text{ not variable, } \theta = \text{mgu}(l, s),\ t[l]\theta \not\succ t[r]\theta \text{ and } t[s]\theta \not\succ u\theta$$

A conclusion of ordered superposition is called an *ordered critical pair*. The conventional notions of superposition and critical pair (without the "ordered" prefix) use the stronger restrictions $l \succ r$ and $t[s] \succ u$. Having made this distinction, we drop the "ordered" qualification for our more general concepts from now on.

As an example, consider the superposition of the associativity axiom (4) with the other list axioms. We obtain the following critical pairs:

(i)	$append(X, Z)$	$=$	$append(X, append(nil, Z))$	from (1) and (4)
(ii)	$append(X, Y)$	$=$	$append(X, append(Y, nil))$	from (1) and (4)
(iii)	$append(Y, Z)$	$=$	$append(nil, append(Y, Z))$	from (2) and (4)
(iv)	$append(A \cdot append(U, Y), Z)$	$=$	$append(A \cdot U, append(Y, Z))$	from (3) and (4)

The superposition inference can be repeatedly used on an equational theory to transform all proofs of the form $t[r]\sigma \leftarrow_E t[l]\sigma \rightarrow_E u\sigma$ to simpler proofs. This process, together with

simplification of the resulting equations, is called (ordered) *completion*. Since proofs of the above form are the only obstacles to confluence, the result of completion is a ground confluent system that defines unique normal forms for all ground terms. See [BDP89, MN90] for a discussion of completion.

In applying equational reasoning to program verification and synthesis, we have a fixed equational program. The equation $l = r$ used in the superposition inference belongs to the fixed program, and need not be treated as a premise of the ifnerence. The specialization of superposition to fixed equational theories is called *linear superposition*:

$$LSup \qquad \frac{t[s] = u}{t[r]\theta = u\theta} \qquad \text{if } s \text{ not variable, } (l = r) \in E,\ \theta = \mathrm{mgu}(l, s),\ t[l]\theta \not\prec t[r]\theta \text{ and } t[s]\theta \not\prec u\theta$$

An important fact that is often overlooked is that the conclusion $t[r]\theta = u\theta$ is, in fact, logically equivalent to $t[s]\theta = u\theta$ given the equational theory E. Thus, the only reason for writing the inference rule in this direction is the specialization involved in the substitution θ. If we consider a sufficient number of θ's to cover all possible ground instances, then we can invert the direction of the inference rule.

Definition 2 *A set of substitutions Θ is said to be* inductively complete *if for every ground substitution σ, there exist $\theta \in \Theta$ and ground substitution τ such that $x\sigma \rightarrow_E^* x\theta\tau$ for all variables x.*

Notice that it is adequate to restrict attention to irreducible σ's in this definition because other substitutions reduce to irreducible ones. Using this notion, we can define a rule for case-based reasoning 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 $\{l_i = r_i\}_i \subseteq E$, $\theta_i = \mathrm{mgu}(l_i, s)$, $t[l_i]\theta_i \not\prec t[r_i]\theta_i$, $t[s]\theta_i \not\prec u\theta_i$ 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 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.*

For example, to prove the associativity property, $append(append(X, Y), Z) = append(X, append(Y, Z))$, as an inductive theorem of (1–3), it is adequate to prove the critical pairs (iii) and (iv). Note that the substitutions $[X \mapsto nil]$, and $[X \mapsto A \cdot U]$ form an inductively complete set. Similarly, to synthesize a program from the specification $rev(X, Y) = append(reverse(X), Y)$, it is adequate to consider the cases:

$$\begin{aligned} rev(nil, Y) &= append(nil, Y) \\ rev(A \cdot U, Y) &= append(append(reverse(U), A \cdot nil), Y) \end{aligned}$$

The two cases are obtained by superposition with (5) and (6) respectively, by overlapping at the subterm $reverse(X)$ of the right hand side of the specification. (Note that the right hand side is the greater side in the specification equation.) Again, the substitutions form an inductively complete set.

The above *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 [Bac88].

Definition 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 exist equation $r = s$ in C and ground substitution τ such that $(t\sigma = u\sigma) \rightarrow_E^+ (r\tau = s\tau)$.*

By $(t\sigma = u\sigma) \rightarrow_E^+ (r\tau = s\tau)$, we mean either $t\sigma \rightarrow_E^+ r\tau$ and $u\sigma \rightarrow_E^+ s\tau$ or $t\sigma \rightarrow_E^+ s\tau$ and $u\sigma \rightarrow_E^+ r\tau$. The general rule for *Cases* uses a cover set of $t = u$ as premises.

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

where $\{r_i = s_i\}_i$ is a cover set of critical pairs of $t = u$. Note that the members of a cover set cannot simply be instances of the conclusion equation. They should incorporate at least one step of reduction using the equations of E. This defines a notion of "progress" for the inference. To formalize this, we define a notion of complexity measures for proofs. With each ground proof step $r\sigma = s\sigma$, we associate a complexity measure $c(r\sigma, s\sigma)$:

$$c(r\sigma, s\sigma) = \begin{cases} (\{r\sigma\}, r, s\sigma) & \text{if } r \succ s \\ (\{s\sigma\}, s, r\sigma) & \text{if } s \succ r \\ (\{r\sigma, s\sigma\}, \bot, \bot) & \text{otherwise} \end{cases}$$

The complexity triples of proof steps are compared by the lexicographic combination of three orderings: multiset extension of $>$ for the first component, the containment ordering[1] $\rhd$ for the second component and $>$ for the third component. The complexity of a proof $t_1 = \ldots = t_n$ is the multiset of the individual proof step complexities, except for the rewrite steps using E which have no cost. Let $>_P$ denote the multiset extension of the above lexicographic ordering.

Lemma 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 theory E whose complexity is strictly less than $c(t\sigma, u\sigma)$.*

Proof: By induction on $c(t\sigma, u\sigma)$. If σ is a reducible substitution with $\sigma \rightarrow_E^+ \sigma'$, use the inductive hypothesis for $t\sigma' = u\sigma'$. If σ is irreducible, there is a critical pair $r_i = s_i$ such that $t\theta \rightarrow_E r_i$, $u\theta \equiv s_i$, $t\theta$ maximal in $t\theta = u\theta$, and θ subsumes σ. (The last property by the definition of cover set.) Let τ be the ground substitution such that $\theta\tau = \sigma$. Then, there is a proof of the form $t\sigma \rightarrow_E r_i\tau = s_i\tau$ with complexity $\{c(r_i\tau, s_i\tau)\}$. By considering the various cases, it may be verified that $\{c(t\sigma, u\sigma)\} >_P \{c(r_i\tau, s_i\tau)\}$. □

An important question is how to test whether a given set of critical pairs is a cover set. Several methods are possible. First, a 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 [Pla85]. 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 three-rule *append* system, $\{nil, A \cdot U\}$ is a test set. This verifies that the critical pairs (iii) and (iv) above form a cover set.

Another method is to use a ground reducibility test. An equation $t = u$ is said to be *ground reducible* if, for every ground instance $t\sigma = u\sigma$, either $t\sigma$ is identical to $u\sigma$ 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 equation in E, then $t\sigma$ is

[1] $s \rhd t$ iff s is an instance of a subterm of t (and $s \neq t$).

covered by a critical pair between the equation in E and $t = u$.) The set of all critical pairs is often too large a cover set. For instance, to prove the associativity property $append(append(X, Y), Z) = append(X, append(Y, Z))$, the cover set need include the critical pairs (iii) and (iv), but not (i) and (ii). The extraneous critical pairs included in the cover set generate other critical pairs so that the proof procedures based on such cover sets may not terminate. A useful optimization that has been suggested in [Fri86, Küc89] is to consider a subterm s of either t or u that is ground reducible. (A term is ground reducible if every ground instance is reducible.) Then superposition at the subterm s is enough to obtain a cover set.

Kapur, *et al.* [KNZ86] suggest another optimization. Note that it is enough to restrict attention to irreducible σ's in the definition of "ground reducible". Suppose $t = u$ is not ground reducible. Then, there is some irreducible ground substitution σ such that $t\sigma$ and $u\sigma$ are distinct and they are both not reducible by E. Thus, whenever $t = u$ is not ground reducible, it equates some pair of irreducible ground terms. They devise an "irreducible ground term" test set to detect this situation. This form of a test set has the property that the equation $t = u$ 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 [JK86, KZ85, BK89].

5 Induction

In synthesizing a program from a specification, we must ensure that the derived program verifies 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 [Red90b] and is based on the "induction by completion" method studied in [Bac88, Der85, Fri86, JK86, KM84, Küc89, Mus80]. The latter is also referred to by "inductionless induction" or "proof by consistency".

An equation $t = u$ is said to be an *inductive consequence* of an equational theory E, written $E \models_{ind} t = u$, if every ground instance $t\sigma = u\sigma$ follows from E. When E is ground confluent (with respect to $>$), this is equivalent to requiring that $t\sigma = u\sigma$ have a rewrite proof using E. Note that 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 $E \models_{ind} t = u$ involves three kinds of steps: we can simplify $t = u$ using the equations in E, we can instantiate it using the *Cases* rule of Section 4, or we can use $t = u$ as an inductive hypothesis in proving one or more of its cases. Notice that, whenever we use the *Cases* rule, we always reduce a maximal side of $t = u$. Since simplification and *Cases* always *reduce* the ground instances of the equation (by the well-founded order $>$), the original equation $t = u$ can be used for simplification of the cases as if it were an "ordinary" equation. This method differs from the conventional induction method 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 [Hoa71]) and fixed point induction [Man74, Sco76].

We make these ideas precise by the following inference procedure. We formulate it in terms of

judgments of the form $H \vdash C$ where C is a set of inductive theorems and H is the set of inductive hypotheses which may be assumed in the proof of C.

Axiom	$\dfrac{}{H \vdash \emptyset}$	
Delete	$\dfrac{H \vdash C}{H \vdash C \cup \{t = t\}}$	
Simplify	$\dfrac{H \vdash C \cup \{t' = u\}}{H \vdash C \cup \{t = u\}}$	if $t \rightarrow_E t'$
Cases	$\dfrac{H \cup \{t = u\} \vdash C \cup C'}{H \vdash C \cup \{t = u\}}$	if C' is a cover set of critical pairs of $t = u$
Induct	$\dfrac{H \vdash C \cup \{t' = u\}}{H \vdash C \cup \{t = u\}}$	if $t \rightarrow_H t'$ by $l = r$ in H and $t \rhd l$
Subsume	$\dfrac{H \vdash C}{H \vdash C \cup \{t[l]\theta = t[r]\theta\}}$	if $(l = r) \in H$
Hypothesize	$\dfrac{H \vdash C \cup \{t = u\}}{H \vdash C}$	

Induct and *Subsume* both apply an inductive hypothesis, but slightly more general treatment is possible for rewrites than for unoriented equational steps. Essentially, a hypothesis can be used for rewriting any number of times, but an unoriented use is possible at most once for each case. The procedure is used by starting with a goal of the form $\emptyset \vdash C_0$ 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 C_0 are all proved and H contains a useful representation of the theorems as well as any lemmas generated in the process.

What if a goal of the form $H \vdash \emptyset$ cannot be obtained? That means that there is an equation $t = u$ in C for which none of the rules *Delete* through *Subsume* are applicable. This means, in particular, that there is no cover set of critical pairs C' for $t = u$. We have already seen that if $t = u$ is ground reducible then the set of all critical pairs with E would be a cover set. So, we conclude that $t = u$ is not ground reducible, *i.e.*, there is a ground instance $t\sigma = u\sigma$ such that $t\sigma$ and $u\sigma$ are distinct normal forms by E. Since E is assumed to be ground confluent, $t\sigma = u\sigma$ does not follow from E and, hence, $t = u$ is not an inductive theorem. Thus, whenever an equation $t = u$ cannot be eliminated from C, we have *disproved* the equation.

Consider proving the associativity property of *append* using the rewrite program (1–3). We start with the goal:

$$\vdash \{append(append(X, Y), Z) \quad = \quad append(X, append(Y, Z))\}$$

Using *Cases*, we can reduce it to

$$\{append(append(X, Y), Z) \quad \rightarrow \quad append(X, append(Y, Z))\} \vdash \left\{ \begin{array}{rcl} append(Y, Z) & = & append(nil, append(Y, Z)), \\ A \cdot append(append(U, Y), Z) & = & append(A \cdot U, append(Y, Z)) \end{array} \right\}$$

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

$$A \cdot append(append(U, Y), Z) \quad = \quad A \cdot append(U, append(Y, Z))$$

Using the inductive hypothesis (by *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 equational theory as a domain fact.

As another example, assume the following program for rev:

$$\begin{array}{rcl} rev(nil, Y) & \rightarrow & Y \\ rev(A \cdot U, Y) & \rightarrow & rev(U, A \cdot Y) \end{array}$$

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

$$rev(X, nil) \quad = \quad reverse(X)$$

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):

$$rev(X, Y) \quad = \quad append(reverse(X), Y)$$

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

$$\{rev(X, Y) \quad \rightarrow \quad append(reverse(X), Y)\} \vdash \left\{ \begin{array}{rcl} Y & = & append(reverse(nil), Y), \\ rev(U, A \cdot Y) & = & append(reverse(A \cdot U), Y), \\ rev(X, nil) & = & reverse(X) \end{array} \right\}$$

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

$$\begin{array}{rcl} rev(U, A \cdot Y) & = & append(append(reverse(U), A \cdot nil), Y) \\ & = & append(reverse(U), A \cdot Y) \end{array}$$

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 an 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 C have proofs using E.

Theorem 6 *Let $H \vdash C$ be a derivable judgment. If all ground instances $r\sigma = s\sigma$ of equations in H have proofs using $E \cup C$ of complexity smaller than $c(r\sigma, s\sigma)$, then all ground instances of C have proofs using E.*

Proof: By induction on the derivation of $H \vdash C$. The proof is trivial for *Axiom*. Suppose

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

is an inference. The plan is to show that the hypothesis of the theorem holds for $H \vdash C$ whenever it holds for $H' \vdash C'$ and that the conclusion holds for C' whenever it holds for C.

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

$$Cases \qquad \frac{H \cup \{t = u\} \vdash C \cup C'}{H \vdash C \cup \{t = u\}}$$

where C' is a cover set of critical pairs of $t = u$. Assume that the hypothesis holds for $H \vdash C \cup \{t = u\}$. To see that it holds for $H \cup \{t = u\} \vdash C \cup C'$, note that ground instances of H already have the required form of proofs using C (by assumption). By Lemma 5, a ground instance of $t = u$ also has a required form of proof using C'.

Next, consider *Induct*:

$$Induct \quad \frac{H \vdash C \cup \{t' = u\}}{H \vdash C \cup \{t = u\}} \quad \text{if } t \rightarrow_H t' \text{ by } l = r \text{ in } H \text{ and } t \rhd l$$

Assume the hypothesis for $H \vdash C \cup \{t = u\}$. We show below that every ground instance $t\sigma = u\sigma$ has a proof using $C \cup \{t' = u\}$ of complexity less than or equal to $c(t\sigma, u\sigma)$. Then, clearly, the hypothesis holds for $H \vdash C \cup \{t' = u\}$ (because all the steps using C are retained and those using $t = u$ are transformed without increase in complexity).

To show the required property for $t\sigma = u\sigma$, we use induction on $c(t\sigma, u\sigma)$. The step $t\sigma = u\sigma$ has a corresponding proof $t\sigma \rightarrow_H t'\sigma = u\sigma$ using the premise of inference. Verify that $c(t\sigma, u\sigma) \geq_P c(t'\sigma, u\sigma)$, considering the various cases in the definition of c. Secondly, suppose the step $t\sigma \rightarrow_H t'\sigma$ is by some $l = r \in H$ with a substitution τ. By assumption, $l\tau = r\tau$ has a proof using $C \cup \{t = u\}$ of complexity complexity less than $c(l\tau, r\tau)$. Verify that $c(t\sigma, u\sigma) \geq_P c(l\tau, r\tau)$, considering the various cases and using the fact $t \rhd l$. So, any use of $t = u$ in the proof of $l\tau = r\tau$ is necessarily of complexity less than $c(t\sigma, u\sigma)$ and, by inductive hypothesis, it has a proof using $C \cup \{t' = u\}$. So, the step $t\sigma \rightarrow_H t'\sigma$ has a corresponding proof using $C \cup \{t' = u\}$ of complexity less than $c(\sigma, u\sigma)$.

Instances of *Subsume* can be verified similarly. □

6 Program Synthesis

Let us return to the problem of program synthesis. To start with, we have an alphabet Σ, an equational axiomatization E and a complete reduction order over T_Σ such that E is ground confluent. The synthesis problem is specified in terms of a new alphabet Σ', an equational specification C, and an extension of the reduction order $>$ to $T_{\Sigma \cup \Sigma'}$. The reduction order must be extended to Σ' in such a way that, for each new symbol $f \in \Sigma'$, f is given higher precedence than the symbols which may appear in its program and lower precedence than the other symbols. For example, considering the synthesis problem for *rev*, given by (1–7), the initial alphabet Σ consists of *reverse*, *append*, $\cdot$ and *nil* (listed in the decreasing order of precedence). The alphabet Σ' consists of *rev* and the precedence order is extended to

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

This indicates that $\cdot$ and *nil* may appear in the program for *rev*, but not *reverse* and *append*.

The program synthesis task is then to derive a program P such that

- P is a consistent enrichment of E, *i.e.*, does not affect the ground equivalences of T_Σ given by E, and
- $E \cup P \models_{ind} C$.

We have already seen, in Section 5, how to verify $E \cup P \models_{ind} C$. To infer P given only E and C, we run the inductive proof procedure with P as an "unknown". The equations $(t = u) \in C$ which cannot be eliminated by any of the inference rules (called "persisting" equations) require knowledge of P. By accepting the set of all persisting equations of C as P, we can trivially satisfy the requirement $E \cup P \models_{ind} C$. Of course, not all such P's satisfy the consistent enrichment condition. We return to that issue below.

Consider synthesizing *rev*. The specification is

$$rev(X, Y) \quad = \quad append(reverse(X), Y)$$

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

$$\begin{aligned} rev(nil, Y) &= append(nil, Y) \\ rev(A \cdot U, Y) &= append(append(reverse(U), A \cdot nil), Y) \end{aligned}$$

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

$$\begin{aligned} rev(nil, Y) &= Y \\ rev(A \cdot U, Y) &= append(reverse(U), A \cdot Y) \end{aligned}$$

Using *Induct*, we can use the inductive hypothesis to reduce the second right hand side to $rev(U, A \cdot Y)$. (Note that this corresponds to a "folding" step in Burstall-Darlington terminology.) No more rules are applicable to these equations. So, the two orientable equations

$$\begin{aligned} rev(nil, Y) &\rightarrow Y \\ rev(A \cdot U, Y) &\rightarrow rev(U, A \cdot Y) \end{aligned}$$

form the candidate program for *rev*.

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

Theorem 7 *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 $t\sigma \notin T_\Sigma$ or $t\sigma, u\sigma \in T_\Sigma$ and $t\sigma$ is reducible by E.*

Proof: The only if direction is immediate. For the if direction, we show by induction on $c(t\sigma, u\sigma)$ that every ground instance $t\sigma = u\sigma$ of an equation in P such that $t\sigma, u\sigma \in T_\Sigma$ 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 proof using $E \cup P$ with complexity less than $c(t\sigma, u\sigma)$. By inductive hypothesis, for each P step in the latter proof, there is 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. The inductive theorems in E do

not affect the ground confluence.) Suppose the completion generates a set $E \cup P_1$. If all equations $t = u$ in P_1, with $t \succ u$, are such that $t \notin T_\Sigma(X)$, then P_1 is an acceptable program. (It is enough to ensure that $t\sigma > u\sigma \Rightarrow t \notin T_\Sigma$, for all ground substitutions σ over T_Σ.) If P_1 contains an equation $t = u$ such that $t, u \in T_\Sigma(X)$ then, we need to verify that $t = u$ is an inductive theorem of E. If $t \in T_\Sigma(X)$ and $u \notin T_\Sigma(X)$, then $t = u$ is a further specification of Σ'. We continue to derive a program for it.

Thus, synthesis is an iterative process. After we find 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 *rev* to the axioms generates no new critical pairs. So, this is indeed the final program for *rev*.

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 [BD77].) We start with the following axioms:

$$f(tip(A)) = A \cdot nil \tag{17}$$
$$f(tip(A) \circ R) = A \cdot f(R) \tag{18}$$
$$f((U \circ V) \circ R) = f(U \circ (V \circ R)) \tag{19}$$
$$nil \approx_L nil = \mathbf{true} \tag{20}$$
$$A \cdot U \approx_L nil = \mathbf{false} \tag{21}$$
$$nil \approx_L B \cdot V = \mathbf{false} \tag{22}$$
$$A \cdot U \approx_L B \cdot V = A \approx B \wedge U \approx_L V \tag{23}$$

(These are used together with the list axioms (1-6) and the propositional axioms in Figure 1.) The fringe equality of trees is then specified by

$$X \approx_F Y = f(X) \approx_L f(Y) \tag{24}$$

We order the function symbols by the precedence

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

Note that all the axioms are orientable left to right using this order.

The synthesis proceeds as follows. We can find a cover set for (24) by considering superposition at the subterm $f(X)$. This gives the cases (shown after possible simplification and induction steps):

$$tip(A) \approx_F Y = A \cdot nil \approx_L f(Y) \tag{25}$$
$$tip(A) \circ R \approx_F Y = A \cdot f(R) \approx_L f(Y) \tag{26}$$
$$(U \circ V) \circ R \approx_F Y = U \circ (V \circ R) \approx_F Y \tag{27}$$

The cases (25) and (26) need further synthesis. This time, we choose $f(Y)$ for superposition. This gives the cases:

$$tip(A) \approx_F tip(A') = A \approx A' \tag{28}$$
$$tip(A) \approx_F tip(A') \circ R' = A \approx A' \wedge nil \approx_L f(R') \tag{29}$$

$$tip(A) \approx_F (U' \circ V') \circ R' = tip(A) \approx_F U' \circ (V' \circ R') \quad (30)$$
$$tip(A) \circ R \approx_F tip(A') = A \approx A' \wedge f(R) \approx_L nil \quad (31)$$
$$tip(A) \circ R \approx_F tip(A') \circ R' = A \approx A' \wedge R \approx_F R' \quad (32)$$
$$tip(A) \circ R \approx_F (U' \circ V') \circ R' = tip(A) \circ R \approx_F U' \circ (V' \circ R') \quad (33)$$

Note that, at this stage, we have *three* inductive hypotheses in the H component of the procedure: (24), (25) and (26). The hypothesis (25) is used in simplifying (30), and (26) is used in simplifying (32) and (33). The only remaining cases which need further work are (29) and (31). Program equations for them can be synthesized using the same process, but we get a clearer program if we postulate the lemmas:

$$nil \approx_L f(X) = \textbf{false} \quad (34)$$
$$f(X) \approx_L nil = \textbf{false} \quad (35)$$

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

$$\begin{aligned}
tip(A) \approx_F tip(A') &\rightarrow A \approx A' \\
tip(A) \approx_F tip(A') \circ R' &\rightarrow \textbf{false} \\
tip(A) \approx_F (U' \circ V') \circ R' &\rightarrow tip(A) \approx_F U' \circ (V' \circ R') \\
tip(A) \circ R \approx_F tip(A') &\rightarrow \textbf{false} \\
tip(A) \circ R \approx_F tip(A') \circ R' &\rightarrow A \approx A' \wedge R \approx_F R' \\
tip(A) \circ R \approx_F (U' \circ V') \circ R' &\rightarrow tip(A) \circ R \approx_F U' \circ (V' \circ R') \\
(U \circ V) \circ R \approx_F Y &\rightarrow U \circ (V \circ R) \approx_F Y
\end{aligned}$$

We also obtain the following inductive theorems as by products:

$$\begin{aligned}
f(X) \approx_L f(Y) &\rightarrow X \approx_F Y \\
A \cdot nil \approx_L f(Y) &\rightarrow tip(A) \approx_F Y \\
A \cdot f(R) \approx_L f(Y) &\rightarrow tip(A) \circ R \approx_F Y \\
nil \approx_L f(X) &\rightarrow \textbf{false} \\
f(X) \approx_L nil &\rightarrow \textbf{false}
\end{aligned}$$

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 [BD77].

7 Generalization

Running the synthesis procedure with domain equations[2]

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

and specification

$$x + x = dx$$

[2] In this section, we usually omit parentheses for unary function symbols.

generates an infinite set of equations:

$$\begin{aligned} d0 &= 0 \\ ssx + x &= dsx \\ ds0 &= ss0 \\ sssssx + x &= dssx \\ dss0 &= ssss0 \\ &\vdots \end{aligned}$$

There is, of course, little one can do with the resultant *infinite* table lookup: $\{ds^i0 = s^{2i}0 \ : \ i \geq 0\}$. What is needed is some way of *guessing* the more general equation $dsx = ssdx$.

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 [BM77]. 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.

For the first step, we overlap the more primitive 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: $d0 \succ 0$, but $ss0 \succ ds0$, $ssss0 \succ dss0$, etc. Using the equations in this direction brings patterns involving d to the fore. From the right-hand sides of

$$\begin{aligned} ds0 &= ss0 \\ dss0 &= ssss0 \\ &\vdots \end{aligned}$$

we get a critical pair

$$\begin{aligned} dss0 &= ssds0 \\ &\vdots \end{aligned}$$

From

$$\begin{aligned} dss0 &= ssss0 \\ dsss0 &= ssssss0 \\ &\vdots \end{aligned}$$

we get

$$\begin{aligned} dsss0 &= ssdss0 \\ &\vdots \end{aligned}$$

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 [Plo70]). 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 $dsx = ssdx$. Applying $dx = x + x$, gives $sx + sx = ss(x + x)$, which simplifies to $s(sx + x) = ss(x + x)$, using the equation $x + sy = s(x + y)$, but no further (in the absence of $sx + y = s(x + y)$). Note that we are assuming $dx \succ x + x$ 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:

$$\begin{aligned} s(sx+x) &= ss(x+x) \\ ss(ssy+y) &= sss(sy+y) \\ &\vdots \end{aligned}$$

Clearly, we need to substitute the (missing) lemma $sx + y = s(x + y)$ for these instances. We employ the same generalization methods as for synthesis (cf. [Jan89, Lan89]). An additional helpful technique is *cancellation*, as used in deduction, for example, in [Sti84]. 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 [HH82]. In the above case, we are free to strip off matching outer s's from the generated hypotheses:

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

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

With this added to the specification, the recursive program

$$\begin{aligned} d0 &= 0 \\ dsx &= ssdx \end{aligned}$$

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

$$\begin{aligned} hdx &= x \\ hsdx &= x \end{aligned}$$

The following sequence of equations is produced:

$$\begin{aligned} h0 &= 0 \\ hs0 &= 0 \\ hss0 &= s0 \\ hsss0 &= s0 \\ hssss0 &= ss0 \\ &\vdots \end{aligned}$$

These equations suggest at least two hypotheses, namely:

$$\begin{aligned} hx &= hsx \\ shx &= hssx \end{aligned}$$

The former generalizes the equations

$$\begin{aligned} h0 &= hs0 \\ hss0 &= hsss0 \end{aligned}$$

but is disproved, since (taking $x = s0$) it implies that $s0 = 0$. The second hypothesis is obtained by looking at different pairs of equations (first and third, second and fourth, etc.) and generalizes the equations

$$\begin{aligned} sh0 &= hss0 \\ shss0 &= hssss0 \end{aligned}$$

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

$$\begin{aligned} h0 &= 0 \\ hs0 &= 0 \\ hssx &= shx \end{aligned}$$

8 Auxiliary Procedures

Most programs require auxiliary procedures, in addition to the specified top-level program. Two heuristics come into play: The first is to abstract a subterm appearing in a program, creating a subprogram to compute it (cf. [Bel91b]). 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. [Red89, Bel91b]).

For example, suppose we have all three equations for addition, and wish to manufacture a program qx for squaring from the following equations for multiplication:

$$\begin{aligned} x \times 0 &= 0 \\ x \times sy &= (x \times y) + x \\ sx \times y &= (x \times y) + y \\ x \times x &= qx \end{aligned}$$

The synthesis procedure will generate the following equations (among others):

$$\begin{aligned} q0 &= 0 \\ s((qx + x) + x) &= qsx \\ sss((qsy + y) + y) &= qssy \end{aligned}$$

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

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

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

$$\begin{aligned} p(x, 0) &= x \\ p(x, sy) &= ssp(x, y) \end{aligned}$$

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

$$qsx = sp(qx, x)$$

With this equation, used from left to right, equations like $sss((qsy + y) + y) = qssy$ 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

$$\begin{array}{rcl} s(qx+(x+x)) & = & qsx \\ sss(qsy+(y+y)) & = & qssy \end{array}$$

suggest an auxiliary function for doubling:

$$x+x \quad = \quad dx$$

That leaves us with the following squaring program:

$$\begin{array}{rcl} q0 & = & 0 \\ qsx & = & s(qx+dx) \end{array}$$

Looking at this program, we note that expanding $qssx$, say, gives $ssss(qx+dx+dx)$. In general, d gets called a quadratic number of times during the computation of $q(x)$. This suggests computing q and d in parallel. Defining

$$\begin{array}{rcl} qx & = & 1^{\mathrm{st}}rx \\ \langle qx, dx \rangle & = & rx \end{array}$$

produces

$$\begin{array}{rcl} r0 & = & \langle 0,0 \rangle \\ \langle s(qx+dx), ssdx \rangle & = & rsx \end{array}$$

Now, suppose we are (conveniently) supplied with the following functions which operate on pairs (higher-order functions would be less ad-hoc):

$$\begin{array}{rcl} s_1\langle x,y \rangle & = & \langle sx, y \rangle \\ s_2\langle x,y \rangle & = & \langle x, sy \rangle \\ a\langle x,y \rangle & = & \langle x+y, y \rangle \end{array}$$

Applying these from right-to-left to simplify the equation for rsx gives the final program:

$$\begin{array}{rcl} qx & = & 1^{\mathrm{st}}rx \\ r0 & = & \langle 0,0 \rangle \\ rsx & = & s_1 s_2 s_2 arx \end{array}$$

Assuming unit cost for addition, this version requires only a linear number of function calls.

9 Discussion

Our approach to synthesis 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* facts for incorporation in developing programs, as well as for forming lemmas needed in inductive proofs.

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.

An interactive program transformation system called Focus has been implemented at University of Illionois based on the techniques presented here. The system incorporates oriented rewrite techniques (a special case of the ordered rewriting techniques considered here) and also several extensions to conditional and first-order reasoning. It has been used to synthesize several interesting examples including some reasonably large programs [Red88, Red90a, Red91].

In this paper, we have 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, [DP88]. Conditional synthesis, however, would necessitate more powerful deductive and inductive methods for handling conditional equations, such as have been investigated in [BR91, KR90, Gan91]. More elaborate generalization methods would also be required.

References

[ACM91] ACM. *Symp. Partial Evaluation and Semantics-Based Program Manipulation.* SIGPLAN Notices, 26(9):1991, 1991.

[Bac88] L. Bachmair. Proof by consistency. In *Symp. on Logic in Comp. Science.* IEEE, 1988.

[BD77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[BDP89] L. Bachmair, N. Dershowitz, and D. A. Plaisted. Completion without failure. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, chapter 1, pages 1–30. Academic Press, 1989.

[BEJ88] D. Bjorner, A. P. Erschov, and N. D. Jones (eds). *Partial Evaluation and Mixed Computation.* North-Holland, 1988.

[Bel91a] F. Bellegarde. Program transformation and rewriting. In R. Book, editor, *Fourth Intern. Conf. on Rewriting Techniques and Applications*, pages 226–239. Springer-Verlag, April 1991. Vol. 488 of *Lect. Notes in Comp. Science.*

[Bel91b] Francois Bellegarde. Program transformation and rewriting. In Ron Book, editor, *Proceedings of the Fourth International Conference on Rewriting Techniques and Applications*, pages 226–239, Como, Italy, April 1991. Vol. 488 of *Lecture Notes in Computer Science*, Springer, Berlin.

[BH84] W. Bibel and K. M. Hörnig. LOPS - A system based on a strategical approach to program synthesis. In A. W. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, chapter 3, pages 69–90. MacMillan Pub. Co., New York, 1984.

[BK86] J. A. Bergstra and Jan Willem Klop. Conditional rewrite rules: Confluency and termination. *J. of Computer and System Sciences*, 32:323–362, 1986.

[BK89] R. Bündgen and W. Küchlin. Computing ground reducibility and inductively complete positions. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, pages 59–75. Springer-Verlag, Berlin, 1989.

[BM77] Robert S. Boyer and J Strother Moore. A lemma driven automatic theorem prover for recursive function theory. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pages 511–519, Cambridge, MA, 1977.

[BR91] F. Bronsard and U. S. Reddy. Conditional rewriting in Focus. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems — Second International CTRS Workshop*, pages 2–13. Springer-Verlag, Berlin, 1991.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice-Hall International, London, 1988.

[Dar81] J. Darlington. The structured description of algorithm derivations. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 221–250. North-Holland, 1981.

[Der82a] N. Dershowitz. Applications of the Knuth–Bendix completion procedure. In *Proc. of the Seminaire d'Informatique Theorique, Paris*, pages 95–111, December 1982.

[Der82b] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.

[Der85] N. Dershowitz. Computing with rewrite systems. *Inf. Control*, 65(2/3):122–157, 1985.

[Der89] N. Dershowitz. Completion and its applications. In *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, pages 31–86. Academic Press, San Diego, 1989.

[Dev90] Y. Deville. *Logic Programming: Systematic Program Development.* Addison-Wesley, Wokingham, 1990.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.

[DK89] N. Dershowitz and S. Kaplan. Rewrite, rewrite, rewrite, rewrite, rewrite, ... In *Sixteenth Ann. ACM Symp. on Princ. of Program. Lang.*, pages 250–259. ACM, January 1989.

[DP88] Nachum Dershowitz and David A. Plaisted. Equational programming. In J. E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11: The logic and acquisition of knowledge*, chapter 2, pages 21–56. Oxford Press, Oxford, 1988. To be reprinted in *Logical Foundations of Machine Intelligence,* Horwood.

[DP90] N. Dershowitz and E. Pinchover. Inductive synthesis of equational programs. In *Eighth National Conf. on Artificial Intelligence*, pages 234–239, Boston, MA, July 1990. AAAI.

[Fea79] M. S. Feather. *A System for Developing Programs by Transformation.* PhD thesis, Univ. of Edinburgh, 1979.

[Fea82] M. S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, 1982.

[FF86] B. Fronhöfer and U. Furbach. Knuth-Bendix completion versus fold/unfold: A comparative study in program synthesis. In C. Rollinger and W. Horn, editors, *Proc. of the Tenth German Workshop on Artificial Intelligence*, pages 289–300, 1986.

[Fri86] L. Fribourg. A strong restriction of the inductive completion procedure. In *Intern. Colloq. Automata, Languages. and Programming*, pages 105–115, July 1986. (Springer Lect. Notes in Comp. Science, Vol. 226).

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

[Gra89] B. Gramlich. Induction theorem proving using refined unfailing completion techniques. Technical Report SR89-14, Universität Kaiserslautern, Germany, 1989.

[HD83] Jieh Hsiang and Nachum Dershowitz. Rewrite methods for clausal and non-clausal theorem proving. In *Proceedings of the Tenth International Colloquium on Automata, Languages and Programming*, pages 331–346, Barcelona, Spain, July 1983. European Association of Theoretical Computer Science. Vol. 154 of *Lecture Notes in Computer Science*, Springer, Berlin.

[HH82] Gérard Huet and Jean-Marie Hullot. Proofs by induction in equational theories with constructors. *J. of Computer and System Sciences*, 25:239–266, 1982.

[HO80] G. Huet and D. C. Oppen. Equations and rewrite rules: A survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, New York, 1980.

[Hoa71] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symp. Semantics of Algorithmic Languages*, volume 188 of *(Lect. Notes in Math.)*, pages 102–116. Springer-Verlag, 1971. (Lect. Notes in Math. Vo. 188).

[Hog76] C. J. Hogger. Derivation of logic programs. *Journal of the ACM*, 23(4), 1976.

[HR86] J. Hsiang and M. Rusinowitch. A new method for establishing refutational completeness in theorem proving. In J. Siekmann, editor, *8th Intern. Conf. on Automated Deduction*, pages 141–152. Springer-Verlag, 1986. (Lect. Notes in Comp. Science).

[HR87] J. Hsiang and M. Rusinowitch. On word problems in equational theories. In T. Ottmann, editor, *14th Intern. Colloq. Automata, Languages and Programming*, pages 54–71. Springer-Verlag, July 1987. (Lect. Notes in Comp. Science Vol. 267).

[Jan89] Klaus P. Jantke. Algorithmic learning from incomplete information: Principles and problems. In J. Dassow and J. Kelemen, editors, *Machines, Languages, and Complexity (Selected Contributions of the 5th International Meeting of Young Computer Scientists, Smolenice, Czechoslovakia, November 1988)*, pages 188–207, 1989. Vol. 381 of *Lecture Notes in Computer Science*, Springer, Berlin.

[JK86] J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in equational theories without constructors. In *Symp. on Logic in Comp. Science*, pages 358–366. IEEE, June 1986.

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

[KB70] D. Knuth and P. Bendix. Simple word problems in Universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[KL80] S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering. Unpublished note, Department of Computer Science, University of Illinois, Urbana, IL, Feb 1980.

[KM84] D. Kapur and D. R. Musser. Proof by consistency. In *Proc. of NSF Workshop on the Rewrite Rule Laboratory, Sep 4-6, 1983*, Schenectady, April 1984. G.E. R&D Center Report GEN 84008.

[KNZ86] D. Kapur, P. Narendran, and H. Zhang. Proof by induction using test sets. In J. Siekmann, editor, *8th Intern. Conf. on Automated Deduction*. Springer-Verlag, 1986. (Lect. Notes in Comp. Science).

[KR87] E. Kounalis and M. Rusinowitch. On word problems in Horn theories. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems*, pages 144–160. Springer-Verlag, Berlin, 1987. (LNCS Vol 308).

[KR90] Emmanuel Kounalis and Michaël Rusinowitch. Inductive reasoning in conditional theories. In M. Okada, editor, *Proceedings of the Second International Workshop on Conditional and Typed Rewriting Systems*, Montreal, Canada, June 1990. *Lecture Notes in Computer Science*, Springer, Berlin; to appear.

[Küc89] W. Küchlin. Inductive completion by ground proof transformation. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 2: Rewriting Techniques, pages 211–245. Academic Press, San Diego, 1989.

[KZ85] E. Kounalis and H. Zhang. A general completeness test for equational specifications. Unpublished report, Centre de Recherche en Informatique de Nancy, Nancy, France, 1985.

[Lan89] Steffen Lange. Towards a set of inference rules for solving divergence in Knuth-Bendix completion. In K. P. Jantke, editor, *Proceedings of the International Workshop on Analogical and Inductive Inference*, pages 304–316, October 1989. Vol. 397 of *Lecture Notes in Computer Science*, Springer, Berlin.

[Man74] Z Manna. *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.

[MN90] Ursula Martin and Tobias Nipkow. Ordered completion. In M. Stickel, editor, *Proceedings of the Tenth International Conference on Automated Deduction*, pages 366–380, Kaiserslautern, West Germany, July 1990. Vol. 449 of *Lecture Notes in Computer Science*, Springer, Berlin.

[Mus80] D. R. Musser. On proving inductive properties of abstract data types. In *ACM Symp. on Princ. of Program. Lang.*, pages 154–162. ACM, 1980.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, January 1980.

[NO91] R. Nieuwenhuis and F. Orejas. Clausal rewriting. In S. Kaplan and M. Okada, editors, *Conditional and Typed Rewriting Systems — Second International CTRS Workshop.* Springer-Verlag, 1991.

[Pla85] D. Plaisted. Semantic confluence tests and completion methods. *Inf. Control*, 65:182–215, 1985.

[Plo70] Gordon Plotkin. Lattice theoretic properties of subsumption. Technical Report MIP-R-77, University of Edinburgh, Edinburgh, Scotland, 1970.

[Red88] U. S. Reddy. Transformational derivation of programs using the Focus system. *SIGSOFT Software Engineering Notes*, 13(5):163–172, Nov 1988. (Proceedings, ACM SIGSOFT/SIGPLAN Softw. Eng. Symp. on Practical Software Development Environments, Also published as SIGPLAN Notices, Feb. 1989).

[Red89] U. S. Reddy. Rewriting techniques for program synthesis. In N. Dershowitz, editor, *Rewriting Techniques and Applications*, pages 388–403. Springer-Verlag, 1989. (LNCS Vol. 355).

[Red90a] U. S. Reddy. Formal methods in transformational derivation of programs. *Software Engineering Notices*, 15(4):104–114, Sep 1990. (Proceedings of the ACM SIGSOFT Workshop on Formal Methods in Software Development).

[Red90b] U. S. Reddy. Term rewriting induction. In M. Stickel, editor, *10th Intern. Conf. on Automated Deduction*, pages 162–177. Springer-Verlag, 1990. (Lecture Notes in Artificial Intelligence, Vol. 449).

[Red91] U. S. Reddy. Design principles for an interactive program derivation system. In M. Lowry and R. D. McCartney, editors, *Automating Software Design*, chapter 18. AAAI Press, 1991.

[RW69] G. Robinson and L. Wos. Paramodulation and theorem-proving in first order theories with equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 135–150. Edinburgh University Press, Edinburgh, Scotland, 1969.

[Sco76] D. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, Sept. 1976.

[Smi90] D. Smith. KIDS - A knowledge-based software development system. In M. Lowry and R. D. McCartney, editors, *Automating Software Design*. AAAI Press, 1990.

[Sti84] Mark E. Stickel. A case study of theorem proving by the Knuth Bendix method discovering that $x^3 = x$ implies ring commutativity. In R. E. Shostak, editor, *Proceedings of the Seventh International Conference on Automated Deduction*, pages 248–259, Napa, CA, May 1984. Vol. 170 of *Lecture Notes in Computer Science*, Springer, Berlin.

[TS84] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Intern. Conf. on Logic Programming*, pages 127–138, 1984.

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