Inductive Synthesis of Equational Programs

Nachum Dershowitz
Department of Computer Science
University of Illinois
Urbana, IL 61801, U.S.A.
nachum@cs.uiuc.edu

Eli Pinchover
Department of Mathematics
and Computer Science
Bar-Ilan University
Ramat-Gan 52100, Israel

Abstract

An equational approach to the synthesis of functional and logic programs is taken. Typically, a target program contains equations that are only true in the standard model of the given domain rules. To synthesize such programs, induction is necessary. We propose heuristics for generalizing from a sequence of deductive consequences. These are combined with rewrite-based methods of inductive proof to derive provably correct programs.

Introduction

Various approaches to the automated synthesis of computer programs have been taken; see (Barr & Feigenbaum 1981–1982, Chap. X). In particular, deductive methods have been used to derive executable programs from formal specifications; early examples of such an approach include (Burstall & Darlington 1977; Manna & Waldinger 1979). Inductive, example-based methods have also been applied to this task; a survey of this approach is (Smith 1980); more recent work includes (Shapiro 1983). In this paper, we employ both deductive and inductive methods of inferring provably correct programs. This work differs from most others in combining syllogistic and heuristic approaches and in using mathematical induction to formally verify hypotheses drawn by inductive inference. It is similar in spirit to some methods used in automated deduction, notably (Boyer & Moore 1977).

In (Dershowitz 1982; Dershowitz 1985a), it was proposed that equations be used both as specification language and as target programming language within a synthesis system. In the equational paradigm, programs are expressed as sets of directed equations, called *rewrite systems*, and are executed using two mechanisms: *rewriting* (reduction) for the functional aspect and *narrowing* (a restricted form of paramodulation) for the "logic" aspect. As inference engine, these papers suggested using the *completion procedure* (Knuth & Bendix 1970). This completion-based approach to synthesis has since been pursued in (Kodratoff & Picard 1983; Perdix 1986; Reddy 1989) and has been compared to the fold/unfold method of (Burstall & Darlington 1977) in (Fronhöfer & Furbach 1986). For a survey of rewriting, see (Dershowitz & Jouannaud 1990); for completion and its applications, see (Dershowitz 1989).

Consider the following toy system S for addition and doubling (d) of natural numbers in unary notation:

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

Such a pattern-directed program is used to compute by replacing instances of a left-hand side pattern (the x and y are variables) with the corresponding instance of the right-hand side. The term $d(s(0))+s(0)$, representing the expression $2*1+1$, may be rewritten by one application of one rule to $(s(0)+s(0))+s(0)$, since the third rule matches the subterm $d(s(0))$, with $x=s(0)$. It could alternatively be reduced to $s(d(s(0))+0)$, applying the second rule to the whole term, with $x = d(s(0))$ and $y=0$. Continuing in any manner (we treat rewrite systems as nondeterministic programs), eventually results in the irreducible term $s(s(s(0)))$, standing for the numeral 3. We say that $s(s(s(0)))$ is a *normal form* of the input term $d(s(0))+s(0)$.

Consider now the following recursive program R, which does not use addition for doubling:

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

System R enjoys four important properties, two intrinsic and two vis-a-vis its specification S: (1) R is *terminating*, i.e. for no input term is an infinite sequence of rewrites possible; (2) R is *ground confluent*, i.e. any variable-free term has at most one normal form; (3) R is *correct* with respect to S, in the sense that terms are only rewritten to terms that are "equal" according to S; (4) R is *complete* with respect to S, in the sense that any two variable-free terms that are equal according to S are also equal according to R. In general, we demand that all synthesized programs meet these requirements.

Section 2 describes the synthesis of deductive consequences of a specification like S. As we will see, deduction alone may produce an infinite program for

S. This leads, in Section 3, to the use of rewriting-based induction techniques to derive the finite program R from a finite subset of the deductive consequences of S. Rewriting-based inductive proofs, called "proofs by consistency" or "inductionless induction", were pioneered by (Musser 1980). It is the use of heuristics for generalizing from a sequence of deductive consequences and then establishing the correctness of the conjectured program by formal inductive techniques that distinguishes this work from previous deductive approaches to program synthesis. A similar approach has independently been taken in (Jantke 1989b).

Deductive Synthesis

The programs we consider in this paper are all in the form of systems of rewrite rules. Rewrite rules are given to the Prolog system as assertions, like `rule(add(X,0),X)` for the rule $x + 0 \rightarrow x$. A simple Prolog interpreter of rewrite programs finds normal forms of input terms. (For things to work right, we take it for granted that "occur-checks" are performed whenever necessary, as can be done in PTTP (Stickel 1986).) With asserted rules corresponding to the program S given earlier, Prolog solves goals like `rewrite(add(d(s(0)),s(0)),Z)`, meaning, "What are the terms `Z` to which the input term `add(d(s(0)),s(0))` rewrites?" The final answer in this case is the normal form `Z = s(s(s(0)))`. Moreover, Prolog can solve goals like `rewrite(add(d(Y),s(0)),s(s(s(0))))`, in which a free ("logic") variable occurs in the term being rewritten, to obtain a solution `s(s(0))` as a value of `Y` that makes `add(d(Y),s(0))` rewrite to `s(s(s(0)))`.

The same program S may be used for subtraction or halving, much like Horn-clause programs may be used to solve for free variables. Instead of using just pattern-matching to locate a potential rewrite, unification is used to make the rewrite possible. To solve the goal equation $d(z) + s(0) = s(s(s(0)))$ for z, one looks for a (nonvariable) subterm of the goal that unifies with a left-hand side of S. (Variables in goals and rules are always treated as disjoint.) After applying the most general unifying substitution to the goal, the enabled step is made. This two-step (unify-rewrite) process is called *narrowing*. The use of narrowing as operational semantics for programming with equations was first suggested in (Dershowitz 1984) and is compared with other approaches in (Reddy 1986).

In our case, the subterm $d(z)$ is an instance of the left-hand side of the third rule, and the goal is rewritten to $(z + z) + s(0)$. (Rewriting is a special case of narrowing in which unification is "one-way" and no substitutions need be made in the goal.) Rewriting twice more, with the second rule followed by the first, gives $s(z + z) = s(s(s(0)))$. Letting $z = 0$ allows the first rule to fire, narrowing the goal to the irreducible, un-narrowable, and unsatisfiable subgoal $s(0) = s(s(s(0)))$. Hence, alternative narrowings must be explored. In this case, we can let $z = s(u)$, instead, yielding the subgoal $s(s(s(u) + u)) = s(s(s(0)))$. Letting $u = 0$, now, gives $s(s(s(0))) = s(s(s(0)))$, the two sides of which are identical. Thus, the solution discovered by narrowing is $z = s(u) = s(0)$.

The basic deductive mechanism we employ in synthesis is *completion* (Knuth & Bendix 1970). *Critical pairs* are equations that are deductive consequences of pairs of rules, formed by unifying one (renamed) left-hand side with a nonvariable subterm of the same or another left-hand-side. The idea in completion is to make new rules out of critical pairs that do not simplify to identity. To generate critical pairs, we use a Prolog predicate `subst(S,L,R,T)` which holds if `T` is the result of applying `rule(L,R)` once at a nonvariable subterm of `S`. The goal solving capabilities of PROLOG allow us to solve for a minimal instance of `S` that makes the rule applicable, yielding—after unification—a critical pair `eq(S,T)`. A terminating system is confluent (a stronger property than ground confluence) if for each critical pair $s = t$, both s and t reduce to the same term v. Some other methods for establishing ground confluence (an undecidable property even of terminating systems) are available; see, for example, (Küchlin 1989).

Completion was programmed as a predicate `kb(Ei,E0)` that applies the basic inference rules described in (Bachmair *et al.* 1986) to `E0` to generate a sequence of sets of equations `E0`, `E1`, etc. One inference rule generates new critical pairs; another reduces them to normal form; a third deletes trivial ones; two inference rules orient critical pairs into uni-directional rewrite rules. Rules are oriented according to a given well-founded ordering on terms, so that applying the rule to any term reduces the term in the ordering, ensuring thereby that the system of generated rules is always terminating (an undecidable property). Completion typically includes additional mechanisms for simplifying rules that have already been generated. (Our implementation does not, and therefore leads to variants of the rules that might be obtained with a full-fledged system like REVE (Lescanne 1983).)

For our purposes, the most useful ordering is the *lexicographic path ordering* (Kamin & Lévy 1980); see (Dershowitz 1987). This ordering $\succ_{lpo}$ is based on a given partial ordering, called a *precedence*, $\succ$ between function symbols. In the induced ordering $\succ_{lpo}$, a term is always greater than each of its subterms, terms with the same leading function symbol are compared lexicographically (from left-to-right, say), and a term with more significant leading symbol needs only be greater than the immediate subterms of a term with less significant leading symbol.

Consider the following specification S:

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

We choose a precedence that ensures that specification symbols (+) are greater than the synthesized functions (d), which in turn are greater than the primitive operations (s,0). The query `kb([],[])` produces the following equations and rules R:

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

The first rule, for instance, is the result of orienting the two normal forms of $0+0$ (in the only direction possible). The latter term is obtained by unifying the left-hand sides of the specification and first rule for addition. For more details of completion-based synthesis, see (Dershowitz 1985b).

We need to show that the two generated rules form a correct and complete system for doubling. To define correctness, we need to make precise what was meant earlier by "equal according to S." There are two notions of equality that are relevant: (a) *deductive equality*, meaning provable by substitution of equals for equals; and (b) *inductive equality*, meaning that all variable-free instances of the equation are deductively equal. To symbolize that every equation in R is a deductive equality with respect to S, we write $S \vdash R$; to symbolize that they are inductive equalities, we write $\mathcal{I}nd(S) \models R$. For example, $S \vdash x+y = y+x$ is *not* true, though for every variable-free instance it is; for example, $S \vdash s(s(0)) + s(0) = s(0) + s(s(0))$. The weaker notion, inductive equality, suffices in programming contexts, so we will say that R is correct with respect to specification S if $\mathcal{I}nd(S) \models R$. When completion is used, correctness is guaranteed in the stronger, deductive sense.

Completeness is similar to correctness, but with the roles of R and S reversed. Actually, we split S into $A \cup D$, where A contains domain information and D expresses properties of the defined functions in terms of domain entities. Then, we require only that $\mathcal{I}nd(A \cup R) \models D$. In the doubling example, A would consist of the three rules for addition and D would be the equation $x + x = d(x)$. The two generated rules for d are complete, since both sides of any ground instance of D (happen to) reduce (under $A \cup R$) to domain values constructed from s and 0. Since R is correct (that is, it follows from $A \cup D$), and D is a conservative extension of A (hence does not equate unequal constructor terms), the two reduced sides of D must be equal in A. So, we have $A \cup R \vdash t + t = d(t)$ for all ground terms t. For a discussion of correctness of synthesized rewrite programs, see (Reddy 1989).

In (Reddy 1989) it is pointed out that full completion is unnecessary and a weaker inference engine suffices; in other words, only a subset of the critical pairs need to be generated for synthesis of ground confluent programs. On the other hand, the additional equations generated by full completion occasionally come in handy. Furthermore, we need not be concerned with the potential completion has of generating an infinite number of irrelevant equations, since—for the purposes of synthesis—completion may be arrested as soon as enough rules R have been generated for completeness.

Ordinary completion will fail if it reaches a point where all critical pairs have been generated, and all equations are nontrivial, in normal form, and unorientable. Unfailing extensions of completion have been designed and perform better; see, for example, (Bachmair *et al.* 1989).

Regardless of which version of completion is employed, it may generate an infinite number of *relevant* program rules. What to do in such a case, is the subject of the next section.

Inductive Synthesis

In (Dershowitz 1985b) and (Reddy 1989), it was argued that—with an appropriate ordering—completion will always generate a program from a given specification. The catch is that the "guaranteed" program may be of infinite length. Indeed, running completion *without* the third addition rule, $s(x) + y \rightarrow s(x+y)$, instead of the desired program, generates an infinite set of rules:

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

There is, of course, little one can do with the resultant *infinite* table lookup: $\{d(s^i(0)) \rightarrow s^{2i}(0) \ : \ i \geq 0\}$. What is needed is some way of guessing the more general rule $d(s(x)) \rightarrow s(s(d(x)))$.

We use two processes to generate hypotheses. The first involves generating critical pairs between *right-hand* sides of rules; the second is a syntactic form of generalization, à la (Boyer & Moore 1977). The intuition is that once we are dissatisfied with the rules, we look for equations between terms containing the defined function symbol, in the hope of discovering a pattern.

For the first step, we generate critical pairs between inverted rules of the current partial program. That is, we use a precedence $+ \succ s, 0 \succ d$, so that patterns involving d are brought to the fore. Given the above rules for d, we get the following equations

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

For the second step, we generate *most specific generalizations* of pairs of equations, by replacing conflicting subterms by a new variable; see (Plotkin 1970). We implemented a predicate `msg(S,T,U,M)` that gives the least general term `U` (i.e. the glb of `S` and `T` in the subsumption lattice) such that `S` and `T` are both instances

of `U`; `M` is the list of conflicts between `S` and `T` used to construct `U`, which is needed to ensure that the same conflicts result in the same new variable. (This process is called "anti-unification" nowadays.)

From the above two equations, `msg` generates the hypothesis $d(s(x)) = s(s(d(x)))$. Given a hypothesized rule (such as this equation oriented from left to right), we must apply the inductive proof method to prove that it is consistent with the domain rules and specification. Once proved, it can be used to reduce (away) special cases already generated.

To prove correctness in the inductive sense, we make use of the following fact: $\mathcal{I}nd(S) \models R$ if and only if $\mathcal{I}nd(S) \models dp(R)$, where $dp(R)$ is the set of all equations obtained by using S to narrow the left side of a rule in R. In proving $dp(R)$, the hypothesis R may be taken for granted. This process may be repeated for each of the equations in $dp(R)$ until only deductive consequences of S remain. In a manner similar to generation of critical pairs, we implemented a predicate `dp(D,E)` that generates equations `E` obtained by narrowing `D`. Thus, to prove an equation `eq(S,T)` we first reduce its two sides to normal form, then consider it proved if it has been shown already, if the two sides are identical, or if it can be oriented into a rule and each equation `E` such that `dp(eq(S,T),E)` is itself provable. This is the essence of the improved proof-by-consistency method in (Fribourg 1989). Refinements of this method include (Bachmair 1988) and (Reddy 1990).

Returning to our example, the hypothesis $d(s(x)) \rightarrow s(s(d(x)))$ is narrowed by $d(x) \rightarrow x + x$, giving $s(x) + s(x) = s(s(d(x)))$. Note that the rule for d is used for verification in the opposite direction of its use for synthesis. This is so that the specification itself is (ground) confluent. The above equation simplifies to $s(s(x) + x) = s(s(x + x))$, using the rule $x + s(y) \rightarrow s(x + y)$, but no further (in the absence of $s(x) + y \rightarrow s(x + y)$). Were this equation provable by deductive means, we would be finished; it isn't, so the inductive proof method continues in the same manner, generating an infinite sequence of oriented hypotheses:

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

Clearly, we need to substitute the (missing) lemma $s(x) + y \rightarrow s(x + y)$ for these instances. We employ the same generalization methods as for synthesis (cf. (Jantke 1989a; Lange 1989)). An additional helpful technique is *cancellation*, as used in deduction, for example, by (Stickel 1984). In particular (Huet & Hullot 1982), 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 there are no rules for c in S. In the above case, we are free to strip off matching outer s's from the generated hypotheses:

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

Using the same `msg` procedure as before leads to the more general hypothesis $s(x) + y = s(x + y)$, exactly what we were looking for.

With this added as a rule to S, the recursive program

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

for d is finally proved correct. The first rule is a deductive consequence of S; the second is an inductive consequence. (Actually, the synthesis procedure would already have suggested the missing equation from the sequence of rules $d(s^i(0)) \rightarrow s^{2i}(0)$, leading to the same result.)

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

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

The following sequence of rules is produced:

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

These rules 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 rules (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

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

When a program R contains symbols not appearing in its specification S, such as symbols for auxiliary functions, we need to replace inductive truth with the

notion of "conservative extension." That is, we require that there exist no terms s and t in the vocabulary of S such that $S \not\vdash s = t$, but $R \cup S \vdash s = t$. One heuristic for introducing auxiliary functions is simply to look for nontrivial subterms that appear (some number of times) in program rules.

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

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

Suppose, further, that we do not wish to allow addition in the synthesized program, so we order the symbols $* \succ + \succ q \succ s, 0$. Completion will generate the following rules (among others):

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

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

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

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

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

Interpolating p in the precedence after q (since it is alright for q to be defined in terms of p) turns all the above rules for q into just:

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

Together, the rules for p and q constitute the desired program.

Discussion

We have shown how—in the equational framework—both deductive and inductive methods may be applied to the task of program synthesis. We have witnessed the need for heuristics that suggest inductive theorems for incorporation in a developing program, as well as for forming the lemmas needed in the inductive proofs. We have been pleasantly surprised by the success of just these few, simple heuristics.

Not all target programs fall under the purview of the automated techniques we have discussed. For example, suppose the specification is

$$\begin{array}{rcl} k(d(x)) & \rightarrow & f(d(x)) \\ k(s(d(x))) & \rightarrow & g(s(d(x))) \end{array}$$

where f and g are primitive operations, and we are given all the facts we have derived about d and h. This is an indirect program for k, requiring narrowing to first solve the goal $y = d(x)$ (or $y = s(d(x))$) for a given number y *before* rewriting $k(y)$ to $f(y)$ (or $g(y)$). Thus, the first rule applies to even y and the second in the odd case. An alternative program, requiring only rewriting would be

$$\begin{array}{rcl} d(h(y)) = y \mid \ k(y) & \rightarrow & f(y) \\ d(h(s(y))) = y \mid \ k(y) & \rightarrow & g(y) \end{array}$$

where the first part is a conditional "guard," which when satisfied for a particular number y allows $k(y)$ to be rewritten. In general, conditional rewrite systems and conditional narrowing provide a better combination of functional and logic programming; see, for instance, (Dershowitz & Plaisted 1988). Conditional synthesis, however, would necessitate more powerful deductive and inductive methods for handling conditional equations, such as have been investigated in (Ganzinger 1987; Kounalis & Rusinowitch 1990). More elaborate generalization methods would also be required, perhaps along the lines of (Kirchner 1989; Lange 1989). This is an area for further research.

References

Bachmair, L. Proof by consistency in equational theories. In *Proceedings of the Third Symposium on Logic in Computer Science*, pp. 228–233, Edinburgh, Scotland, July 1988.

Bachmair, L., Dershowitz, N., and Hsiang, J. Orderings for equational proofs. In *Proceedings of the Symposium on Logic in Computer Science*, pp. 346–357, Cambridge, MA, June 1986.

Bachmair, L., Dershowitz, N., and Plaisted, D. A. Completion without failure. In Ait-Kaci, H. and Nivat, M., eds., *Resolution of Equations in Algebraic Structures*, vol. 2: Rewriting Techniques, chap. 1, pp. 1–30, Academic Press, New York, 1989.

Barr, A. and Feigenbaum, E. A., eds. *The Handbook of Artificial Intelligence*. William Kaufmann, 1981–1982. Three volumes.

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

Burstall, R. M. and Darlington, J. A transformation system for developing recursive programs. *J. of the Association for Computing Machinery*, 24(1):44–67, January 1977.

Dershowitz, N. Applications of the Knuth-Bendix completion procedure. In *Proceedings of the Seminaire d'Informatique Theorique*, pp. 95–111, Paris, France, December 1982.

Dershowitz, N. Equations as programming language. In *Proceedings of the Fourth Jerusalem Conference on Information Technology*, pp. 114–124, IEEE Computer Society, Jerusalem, Israel, May 1984.

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

Dershowitz, N. Synthesis by completion. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 208–214, Los Angeles, CA, August 1985.

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

Dershowitz, N. Completion and its applications. In Ait-Kaci, H. and Nivat, M., eds., *Resolution of Equations in Algebraic Structures*, vol. 2: Rewriting Techniques, chap. 2, pp. 31–86, Academic Press, New York, 1989.

Dershowitz, N. and Jouannaud, J. Rewrite systems. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science B: Formal Methods and Semantics*, chap. 6, North-Holland, Amsterdam, 1990. In press; available as Rapport 478, LRI, Univ. Paris-Sud, France.

Dershowitz, N. and Plaisted, D. A. Equational programming. In Hayes, J. E., Michie, D., and Richards, J., eds., *Machine Intelligence 11: The logic and acquisition of knowledge*, chap. 2, pp. 21–56, Oxford Press, Oxford, 1988.

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

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

Ganzinger, H. A completion procedure for conditional equations. In Kaplan, S. and Jouannaud, J., eds., *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, pp. 62–83, Orsay, France, July 1987. Vol. 308 of *Lecture Notes in Computer Science*, Springer, Berlin (1988).

Huet, G. and Hullot, J. Proofs by induction in equational theories with constructors. *J. of Computer and System Sciences*, 25:239–266, 1982.

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

Jantke, K. P. *Inductive program synthesis by problem proving and term rewriting.* Technical Report, Humbolt Univ. Berlin, Berlin, 1989.

Kamin, S. and Lévy, J. J. *Two generalizations of the recursive path ordering.* Unpublished note, Department of Computer Science, University of Illinois, Urbana, IL, February 1980.

Kirchner, H. Schematization of infinite sets of rewrite rules generated by divergent completion processes. *Theoretical Computer Science*, 67(2,3):303–332, October 1989.

Knuth, D. E. and Bendix, P. B. Simple word problems in universal algebras. In Leech, J., ed., *Computational Problems in Abstract Algebra*, pp. 263–297, Pergamon Press, Oxford, U. K., 1970. Reprinted in *Automation of Reasoning 2*, Springer, Berlin, pp. 342-376 (1983).

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

Kounalis, E. and Rusinowitch, M. Inductive reasoning in conditional theories. In Okada, M., ed., *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.

Küchlin, W. Inductive completion by ground proof transformation. In Ait-Kaci, H. and Nivat, M., eds., *Resolution of Equations in Algebraic Structures*, vol. 2: Rewriting Techniques, pp. 211–244, Academic Press, New York, 1989.

Lange, S. 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*, pp. 304–316, October 1989. Vol. 397 of *Lecture Notes in Computer Science*, Springer, Berlin.

Lescanne, P. Computer experiments with the reve term rewriting system generator. In *Proceedings of the Tenth ACM Symposium on Principles of Programming Languages*, pp. 99–108, Austin, TX, January 1983.

Manna, Z. and Waldinger, R. J. Synthesis: Dreams → programs. *IEEE Transactions on Software Engineering*, SE-5(4):294–328, July 1979.

Musser, D. R. On proving inductive properties of abstract data types. In *Proceedings of the Seventh ACM Symposium on Principles of Programming Languages*, pp. 154–162, Las Vegas, NV, 1980.

Perdix, H. Program synthesis from specifications. In Jorrand, P. and Sgurev, V., eds., *AIMSA*, pp. 13–21, North-Holland, 1986.

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

Reddy, U. S. On the relationship between logic and functional languages. In DeGroot, D. and Lindstrom, G., eds., *Logic Programming: Functions, Relations, and Equations*, pp. 3–36, Prentice-Hall, Englewood Cliffs, NJ, 1986.

Reddy, U. S. Rewriting techniques for program synthesis. In Dershowitz, N., ed., *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pp. 388–403, Chapel Hill, NC, April 1989. Vol. 355 of *Lecture Notes in Computer Science*, Springer, Berlin.

Reddy, U. S. Term rewriting induction. In Stickel, M., ed., *Proceedings of the Ninth International Conference on Automated Deduction*, Kaiserslautern, West Germany, July 1990. *Lecture Notes in Computer Science*, Springer, Berlin; to appear.

Shapiro, E. Y. *Algorithmic Program Debugging.* MIT Press, Cambridge, MA, 1983.

Smith, D. R. A survey of synthesis of Lisp programs from examples. In *International Workshop on Program Construction*, Bonas, France, September 1980.

Stickel, M. E. 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*, pp. 248–259, Napa, CA, May 1984. Vol. 170 of *Lecture Notes in Computer Science*, Springer, Berlin.

Stickel, M. E. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. In Siekmann, J. H., ed., *Proceedings of the Eighth International Conference on Automated Deduction*, pp. 573–587, Oxford, England, July 1986. Vol. 230 of *Lecture Notes in Computer Science*, Springer, Berlin.