SYNTHESIS BY COMPLETION*

Nachum Dershowitz
Department of Computer Science
University of Illinois
Urbana, IL 61801
U.S.A.

ABSTRACT

The Knuth-Bendix completion procedure was introduced as a means of deriving canonical term-rewriting systems to serve as decision procedures for given equational theories. The procedure generates new rewrite rules to resolve ambiguities resulting from existing rules that overlap. We propose using this procedure to synthesize logic programs, as well as functional programs, from specifications and domain knowledge expressed as equivalence-preserving rewrite rules. An implementation is underway.

1. INTRODUCTION

A (first-order) *functional (applicative) program* is a set of directed equations, used to compute by replacing instances of left-hand sides with the value of the right-hand sides. For example, the following is a program for concatenating two lists of elements:

Functional List Append
$append(U,V) \Leftarrow$ **if** $U=nil$ **then** V **else** $car(U)\cdot append(cdr(U),V)$

where <= has the declarative meaning "is equal" *(nil* is the empty list and • denotes the *cons* function). Given the term

$$append(a\cdot b\cdot nil, c\cdot d\cdot e\cdot nil),$$

it will compute the result of appending the two elements *a* and 6 in the list *a.b.nil* [with parentheses, that should be *a(bnil)]* to the front of the list *c.d.e.nil.* *A logic program* [Kowalski-74] is a set of Horn clauses used as a pattern-directed program that searches for output terms that satisfy a given goal for given input terms. In this paradigm, the *append* program would be expressed in two statements:

Logical List Append
$append(nil,V,V)$
$append(A\cdot U,V,A\cdot W) \quad :- \quad append(U,V,W)$

*This research was supported in part by the National Science Foundation under Grant MCS 83-07755 and in part by the Israeli Ministry of Defense while the author was at the Department of Mathematics and Computer Science, Bar-Han University, Ramat-Gan, Israel.

where :— has the declarative meaning "is implied by". Given a goal

append (a b nil ,c d e nil ,Z),

this program computes the appended list *Z=a.b.c.d.e.nil.* Throughout this paper we follow the convention of using lower case for constants and upper case for free (universally quantified) variables.

A *rewrite system* is a set of directed equations (or equivalences) used as a *nondeterministic* pattern-directed program that returns as output a simplified term equal to a given input term. (See [Huet-Oppen-80] for a survey.) For example, the following is a three-rule rewrite system for *append:*

List Append		
$append(nil,V)$	$\rightarrow$	V
$append(U,nil)$	$\rightarrow$	U
$append(A\cdot U,V)$	$\rightarrow$	$A\cdot append(U,V)$

Rules may be applied in any order to any matching subterm until no further applications are possible. Thus, applying the rules to the term

append {a nil,nil),

one gets either the rewrite sequence

append (a nil ,nil) $\Rightarrow$ *a append (nil ,nil)* $\Rightarrow$ *a nil,*

or (using only the second rule) simply

append (a nil ,nil) $\Rightarrow$ *a nil.*

Such pattern-directed functional programming is available in SASL [Turner-79], HOPE [Burstall-MacQueen-Sannella-80], OBJ2 [Futatsugi,et al.-84], rewrite languages [Hoffmann-O'Donnell-82], and PLANNER-like languages.

In this paper, we describe how the Knuth-Bendix *completion procedure* [Knuth-Bendix-70] may be used to generate programs from their (equational) specifications. The completion procedure was originally suggested as a means of generating rewrite systems that can be used to decide the validity of identities in equational theories. Given a finite set $\mathcal{E}$ of equations and a monotonic, well-founded ordering $\succ$ on terms, the procedure generates

new rules that follow from . Each rule $l \rightarrow r$ generated by the procedure is a *reduction* with respect to $\succ$, i.e. $l \succ l'$ whenever $l \Rightarrow l'$ (*t* rewrites to /') by a single application of the rule, and each rule is *sound* for $\mathcal{E}$, i.e. $l = r$ follows from £ by "replacing equals by equals." Implementations of the completion procedure include REVE [Lescanne-83], RRL at GE [Kapur-Sivakumar-83], and FORMEL at 1NRIA [Hullot-80].

A rewrite system synthesized by the completion procedure (and not containing nested defined symbols on left-hand sides) can be translated into a terminating functional program in a straightforward manner—with the pattern-directed invocation eliminated, if desired. For example, the above rewrite system for *append* could translate to

$$\begin{aligned} append(U,V) \quad \Leftarrow \quad & \textbf{if } U = nil \textbf{ then } V \\ & \textbf{else if } V = nil \textbf{ then } U \\ & \textbf{else if } listp(U) \textbf{ then} \\ & \quad car(U) \cdot append(cdr(U), V) \\ & \textbf{else } error. \end{aligned}$$

Function definitions can be translated into Horn clauses in the standard manner [by encoding a rule like $f(X) \rightarrow g(h(X))$ as a clause *f(X)=Z :- h(X)=Y, g(Y)=Z].* Logic programs differ from functional ones in that output variables in a goal are instantiated by unification with the head of a clause. Rewrite rules equating conjunctions correspond to Horn clauses which may be used to solve goals in that way. For example, a rule of the form

$$P \ \& \ Q \ \& \ R \ \rightarrow \ Q \ \& \ R \ \& \ S$$

corresponds to the two Horn clauses (one for each conjunct that appears on only one side):

$$\begin{aligned} P \ &:- \ Q, R, S \\ S \ &:- \ P, Q, R; \end{aligned}$$

a rule *P-->true* corresponds to the assertion *P.* In this way, the above logic-program for *append* follows from the two-rule system:

$$\begin{aligned} append(nil,V,V) \quad & \rightarrow \quad true \\ append(A \cdot U, V, A \cdot W) \ \& \ & append(U,V,W) \\ & \rightarrow \ append(U,V,W). \end{aligned}$$

Note that, in general, a logic program formed from a rewrite system in this way is not guaranteed to terminate when backtracking is employed in the search for solutions to a given goal.

A rewrite system is *complete* (as a functional or logic program) if the patterns on the left-hand sides of its rules cover all possible (variable-free) input values. One can determine that a program is complete, and all cases are covered, using methods described in [Thiel-84, Dershowitz-85, Kounalis-Zhang-85].

Not all rewrite systems correspond to functional or logic programs. Rewrite rules with nested defined symbols on their left-hand side do not have a straightforward analogue in all functional languages. And equivalences between formulae other than conjunctions do not have Horn clause counterparts. [Dershowitz-85] describes how a restricted form of completion, called "narrowing," can be used as an interpreter for logic programs expressed as rewrite rules that are not necessarily Horn clauses.

2. COMPLETION

Before we present the completion procedure, we need to define the notion of "critical pair." Let $L \rightarrow R$ and $L' \rightarrow R'$ be two (not necessarily different) rules in a system $\mathcal{R}$ whose variables have been renamed, if necessary, so that they are distinct. We say that L' *overlaps* L, if the latter has a subterm that unifies with the former. More formally, L' overlaps L if L contains a (nonvariable) subterm S embedded in some context C– to indicate this we write $L = C[S]$ such that there is a (most general) substitution σ for the variables of the two rules for which $S\sigma = L'\sigma$. If L overlaps L', then the overlapped term $L\sigma$ (L after making the substitution σ) can be rewritten to either $R\sigma$ or $C[R']\sigma$. These two terms, $R\sigma$ and $C[R']\sigma$, are called a *critical pair* of $\mathcal{R}$.

The completion procedure takes as input a finite set $\mathcal{E}$ of equations and a program to compute a monotonic well-founded ordering $\succ$. (By "monotonic" we mean that $C[T]\sigma \succ C[T']\sigma$ whenever $T \succ T'$, for any context C, substitution σ, and terms T and T'.) The procedure then generates new rules, each of which is a sound reduction:

Completion Procedure

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

[1] *Choose an equation $M = N$ (or $N = M$) from $\mathcal{E}$ such that $M \succ N$. If none exists, backtrack and undo previous choice. If all choices have been exhausted, terminate with failure* (abort).

[2] *Add the rule $M \rightarrow N$ to $\mathcal{R}$.*

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

[4] *Add to $\mathcal{E}$ all critical pairs formed from $\mathcal{R}$ using $M \rightarrow N$.*

[5] *Remove all the old rules from $\mathcal{R}$ whose left-hand side contains an instance of M.*

[6] *Use $\mathcal{R}$ to reduce both sides of equations in $\mathcal{E}$ to their normal forms. Remove any equation that reduces to identity.*

In general, the completion procedure may go on generating an infinite number of new rules. But, sometimes, despite the backtracking in Step [1], a particular choice of (partial) ordering $\succ$ forces the whole procedure to fail with all equations unorientable. On the other hand, as long as the completion procedure does not terminate in failure, for any equation $M=N$ that follows (logically) from the given rules R (considered as equations), the procedure will eventually generate enough rules for M and N to reduce to the identical term. For this to be the case, the procedure must execute *fairly,* by which we mean that no orientable equation remains in £ forever.

The completion procedure has been extended [Peterson-Stickel-81, Fages-83] to rewrite systems that contain associative-commutative function symbols (possibly with identity element) by considering a somewhat extended notion of critical pair and a unification algorithm for associative-commutative symbols. If / is an associative-commutative symbol, then critical pairs are also formed from overlapped terms of the form $f(L'\sigma, X\sigma) = L\sigma$, where X is a new variable, and to which two rules $L \to R$ and $L' \to R'$ (both with associative-commutative outermost symbols) may be applied.

3. SYNTHESIS

Like other theorem-proving methods, completion can be applied to the task of automatic program synthesis from specifications. The completion procedure itself does the "folding" (that is, the introduction of recursive calk) based upon the axiomatization of the problem domain. Other work on the synthesis of recursive programs includes [Burstall-Darlington-77, Manna-Waldinger-80, Clark-81, Hogger-81]. Specifications are expressed as rewrite rules, i.e. as directed equations or equivalences. The importance of using equivalences in specifications, rather than implications has been pointed out in [Kowalski-79, Hogger-81], and others. Heuristic aspects of the synthesis problem are not addressed in this paper.

Assume that we wish to synthesize a program for some function (or predicate) /, and are given an axiomatization £ of the problem domain. We can start the completion procedure off with $\mathcal{E}$ and run it until a program R is generated that computes /. (In our examples, we will skip the first stage of completion, starting off with already oriented rules for £, and considering critical pairs as necessary.) The monotonic well-founded ordering supplied to the completion procedure should ensure that terms containing "specification" symbols are greater than corresponding terms containing the defined goal symbol, which in turn should be greater than the constant *true.* The choice of ordering guides the synthesis and will affect the program derived. Given the "right" ordering, the procedure will find a program, if it does not abort on account of inability to orient any equation into a rule, and if a program exists that does not require auxiliary definitions. (See [Dershowitz-85].) When auxiliary procedures are needed, their definition may be supplied by the user.

As a simple example of the use of completion to "fold" and "unfold," we synthesize an efficient program to reverse a list, given the naive version that uses *append* (cf. the same problem in [Burstall-Darlington-77]). The naive program is

$$reverse\ (nil) \to nil \quad (1)$$

$$reverse(AX) \to append\ (reverse\ (X),\ A\ nil) \quad (2)$$

The synthesis requires a definition of the auxiliary function *rev:*

$$append(reverse\ (Y)_i Z) \to rev(Y,Z), \quad (3)$$

the *append* system of Section 1:

$$append\ (nil\ ,V) \to V \quad (4)$$

$$append\ (U,nil) \to U \quad (5)$$

$$append(AUy) \to A \cdot append\ (U\ ,V), \quad (6)$$

and the theorem (associativity of *append):*

$$append\ (append\ (X\ ,Y),Z) \to append(X,\ append(Y,Z)) \quad (7)$$

Completion proceeds as follows: The left-hand sides of rules (3) and (5) can be unified by letting Z in (3) be *nil* and V in (5) be *reverse(Y).* That generates the critical pair

$$reverse\ (Y) = rev\ (Y,nil).$$

Since we want *reverse* to use *rev*, the ordering supplied to the completion procedure should make anything containing *reverse* bigger than a term that does not. That way, the above equation is oriented into a rule

$$reverse\ (Y) \to rev\ (Y\ .nil). \quad (8)$$

The next step is to overlap the naive *reverse* program with the definition of *rev.* Unifying the left-hand side of (1) with the subterm *reverse (Y)* of (3), by letting y = n i l , gives rise to the critical pair

$$append\ (nil,Z) = rev\ (nil,Z),$$

the left side of which reduces, using (4), to just Z. There being only one possible way to orient the equation $Z=rev(nil,Z)$ into a (terminating) reduction, that pair results in the rule

$$rev\ (nil,Z) \to Z. \quad (9)$$

Overlapping (3) now with the left-hand side of (2), letting $Y=A.X$ gives the critical pair

$$append\ (append\ (reverse\ (X),\ A\ nil),Z) = rev(AX,Z).$$

Associativity of *append* (7) comes in here, rewriting the

left side to *append (reverse (X), append (A nil,Z))*, which becomes *rev(X,AZ)* by applying (6), (4), and (3). With an appropriate ordering (one that begins by looking at the first argument of *rev,* on which the intended program recurs), that gives the rule

$$rev(AX,Z) \rightarrow rev(X,AZ). \quad (10)$$

The three rules generated by completion, viz. (8-10), serve as a functional program for *reverse,* one that does not itself use *append*:

List Reversal		
$reverse(Y)$	$\rightarrow$	$rev(Y,nil)$
$rev(nil,Z)$	$\rightarrow$	Z
$rev(A\cdot X,Z)$	$\rightarrow$	$rev(X,A\cdot Z)$

This program is complete, since Y covers all lists for *reverse* and *(nil ,Z)* and *(A X ,Z)* cover all possible pairs of lists for *rev.* The program is terminating, since completion used a well-founded ordering to the rules.

4. LOGIC PROGRAMS

Suppose that we are given the following definition of multiplication (for natural numbers):

$$M\times 0 \rightarrow 0 \quad (1)$$
$$M\times(N+1) \rightarrow M\times N+M, \quad (2)$$

where + and x are associative and commutative (with identities 0 and 1, respectively), and that we wish to synthesize a program for integer division. We will also need the following four facts, expressed as simplification rules:

$$U \,\&\, true \rightarrow U \quad (3)$$
$$U=U \rightarrow true \quad (4)$$
$$U\leq U+Z \rightarrow true \quad (5)$$
$$U+W=V+W \rightarrow U=V. \quad (6)$$

Positive integers are represented in unary, as sums of ones. (Associativity and commutativity are needed so that *U + W,* for example, can be matched with 1 + 1 + 1 + 1, with U = V = I + I.)

The completion procedure starts off with the above rules (1-6) and the specification

$$R\leq Y \;\&\; X=(Y+1)\times Q+R \rightarrow div(X,Y+1,Q,R). \quad (7)$$

The procedure is also given a recursive path ordering (see [Dershowitz-82]) in which function symbols are ordered (from "heavier" to "lighter"): $\&$, $=$, $<$, *div*, +, 1,0, *true.* The synthesis proceeds is as follows:

By overlapping (1) $M\times 0\rightarrow 0$ on the specification (7) (unifying y + 1 with *M* and *Q* with 0), we get

$$R\leq Y \;\&\; X=0+R \rightarrow div(X,Y+1,0,R). \quad (8)$$

By overlapping the fact (4) $U=U\rightarrow true$ on (8) (unifying X, U, and $0+R$ -the unification algorithm takes the identity $0+R=R$ into account), and simplifying the resulting left side $X\leq Y \,\&\, true$ to $X\leq Y$ via (3), we get

$$X\leq Y \rightarrow div(X,Y+1,0,X). \quad (9)$$

Next, by overlapping the fact (5) $U\leq U+Z\rightarrow true$ on (9) (unifying U with X and Y with $X+Z$), we obtain

$$div(X,X+Z+1,0,X) \rightarrow true. \quad (10)$$

Now, by using (9) to reduce $R\leq Y$ in the specification (7) to $div(R,Y+1,0,R)$, and multiplying out $(Y+1)\times Q$ using (2), we get

$$div(R,Y+1,0,R) \;\&\; X=Y\times Q+Q+R \rightarrow div(X,Y+1,Q,R). \quad (11)$$

Overlapping (11) with (2) $M\times(N+1)\rightarrow M\times N+M$ (letting $Q=N+1$) yields

$$div(R,Y+1,0,R) \;\&\; X=Y\times N+Y+N+1+R \rightarrow div(X,Y+1,N+1,R). \quad (12)$$

Finally, by overlapping the fact (6) $U+W=V+W\rightarrow U=V$ on (12) (using associativity and commutativity of + to let $W=Y+1$, $V=Y\times N+N+R$, and $X=U+Y+1$), we obtain the critical pair

$$div(R,Y+1,0,R) \;\&\; U=Y\times N+N+R = div(U+Y+1,Y+1,N+1,R).$$

The left side reduces by (11) to $div(U,Y+1,N,R)$, yielding the rule

$$div(U+Y+1,Y+1,N+1,R)\rightarrow div(U,Y+1,N,R). \quad (13)$$

Rules (10) and (13) together are a complete program. Renaming variables and putting into Horn clause form, we have the following:

Integer Division
$div(X,X+Z+1,0,X)$
$div(X+Y+1,Y+1,Q+1,R) \text{ :- } div(X,Y+1,Q,R)$

The second rule is the recursive case (denominator not greater than numerator); the first is the base case (denominator is greater). The program is complete, since, for any nonnegative integer a and positive integer b, the input pair (x,y) matches $(X,X+Z+1)$ when $a<b$ and $(X+Y+1,Y+1)$ when $a\geq b$ (X is taken to be 0 for $a=b$). Executing this program as it stands requires associative-commutative unification for +, with 0 as its identity element; getting around that would require additional programs for unary subtraction and comparison.

5, FORWARD REASONING

In this section, we synthesize a rewrite system that can be used to search for an integral position P such that the input value X lies between $f(P)$ and $f(P+1)$, for monotonically nondecreasing function /. The resultant logic program computes by "forward reasoning" [Kowalski-79], from facts towards the goal.

The following propositional calculus system (cf. [Hsiang-Dershowitz-83]) can provide additional logical capability for specifications:

Propositional Calculus		
$\neg U$	$\rightarrow$	$U=false$
$U \vee V$	$\rightarrow$	$(U \,\&\, V)=U=V$
$U \supset V$	$\rightarrow$	$(U \,\&\, V)=U$
$U \,\&\, true$	$\rightarrow$	U
$U \,\&\, false$	$\rightarrow$	$false$
$U \,\&\, U$	$\rightarrow$	U
$U=true$	$\rightarrow$	U
$U=U$	$\rightarrow$	$true$
$(U=V) \,\&\, W$	$\rightarrow$	$(U \,\&\, W)=(V \,\&\, W)=W$

where $\neg$ is "not," $\&$ is "and," $\vee$ is "inclusive-or," $=$ is "equivalence," and $\supset$ is "implies." Both $\&$ and $=$ are implicitly associative and commutative. That means, for example, that the rule $U \,\&\, U \rightarrow U$ applied to $(p \,\&\, q) \,\&\, p$ yields $p \,\&\, q$. Since these two functions are associative, there is no significance to their parenthesization, and terms are accordingly "flattened" by removing embeddings of associative functions symbols, e.g. $(p \,\&\, q) \,\&\, p$ is written $p \,\&\, q \,\&\, p$. The above system simplifies any two equivalent propositional formulae to the same unique irreducible term.

The completion procedure starts off with the output specification

$$X \geq f(P) \;\&\; X < f(P+1) \;\rightarrow\; search(P) \qquad (1)$$

(X lies between P and $P+1$), input specification

$$f(U+V) \geq f(U) \;\rightarrow\; true \qquad (2)$$
$$X \geq f(A) \;\rightarrow\; true \qquad (3)$$
$$X < f(A+N) \;\rightarrow\; true \qquad (4)$$

(f is monotonically nondecreasing and X lies between A and $A+N$), and facts about transitivity of inequality:

$$U<V \;\&\; W \geq V \;\&\; U<W \;\rightarrow\; U<V \;\&\; W \geq V \qquad (5)$$
$$V \geq U \;\&\; W \geq V \;\&\; W \geq U \;\rightarrow\; V \geq U \;\&\; W \geq V \qquad (6)$$

(Any implication of the form $U \supset V$ can be expressed by a rule $U \,\&\, V \rightarrow U$.) The synthesis requires the programmer to introduce a generalization $pos(P,V)$ of $search(P)$, and to introduce halving of integers, to guide the synthesis to binary search, rather than linear search. (The ordering on symbols has *pos* between the specification symbols and the goal symbol *search*.)

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

$$X \geq f(P) \;\&\; X < f(P+Y) \;\rightarrow\; pos(P,Y), \qquad (7)$$

replacing 1 with Y, and meaning that X lies between P and $P+Y$. This generates the rule

$$pos(P,1) \;\rightarrow\; search(P), \qquad (8)$$

in place of (1), as well as

$$X < f(A+Y) \;\rightarrow\; pos(A,Y) \qquad (9)$$
$$pos(A,N) \;\rightarrow\; true, \qquad (10)$$

by overlapping (7) with (3) and (4). Transitivity (5,6), together with (2), generate

$$W < f(U) \;\&\; W < f(U+V) \;\rightarrow\; W < f(U) \qquad (11)$$
$$W \geq f(U) \;\&\; W \geq f(U+V) \;\rightarrow\; W \geq f(U+V). \qquad (12)$$

In turn, these and (7) generate

$$pos(P,V+W) \;\&\; X < f(P+V) \rightarrow pos(P,W) \qquad (13)$$
$$pos(P,V+W) \;\&\; X \geq f(P+V) \rightarrow pos(P+V,W). \qquad (14)$$

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

Nondeterministic Search		
$search(P)$	:-	$pos(P,1)$
$pos(A,N)$		
$pos(P,V)$	:-	$pos(P,V+W) \;\&\; X < f(P+V)$
$pos(P+V,W)$	:-	$pos(P,V+W) \;\&\; X \geq f(P+V)$

Note that the left-hand side of a rewrite rule becomes the list of premisses of a clause used for forward-reasoning.

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

$$(U+2)+((U+1)+2) \;\rightarrow\; U. \qquad (15)$$

This generates

$$pos(P,Y) \;\&\; X < f(P+(Y+2)) \rightarrow pos(P,Y+2) \qquad (16)$$
$$pos(P,Y) \;\&\; X \geq f(P+(Y+2)) \rightarrow pos(P+(Y+2),(Y+1)+2) \qquad (17)$$

from (13) and (14). Rules (8,10,16,17) constitute the binary-search program:

Binary Search		
$search(P)$	:-	$pos(P,1)$
$pos(A,N)$		
$pos(P,Y+2)$	:-	$pos(P,Y) \;\&\; X < f(P+(Y+2))$
$pos(P+(Y+2),(Y+1)+2)$	:-	$pos(P,Y) \;\&\; X \geq f(P+(Y+2))$

Given values for *X, A,* and *N,* along with programs for / and +, this program computes *P* such that *search (P)* holds. Starting from the axiom *pos(A,N),* it tests inequalities to add new facts of the form *pos(P,Y),* until the fact *pos(P,1),* and its consequence *search(P),* are generated.

6. AUXILIARY PROCEDURES

The above approach requires that a program be specified equationally. That means that it may be necessary to give recursive definitions of predicates appearing in specifications. (There is a comparable need of definitions for verification purposes in [Boyer-Moore-79].) For example, the following insertion-sort program

Insertion Sort
$sorted(A \cdot X, Z)$ & $sorted(X, Y)$
→ $sorted(X, Y)$ & $inserted(A, Y, Z)$
$sorted(nil, nil)$ → $true$

requires the additional rules

Linear Insertion
$inserted(A, B \cdot X, B \cdot Z)$ & $B \leq A$ & $sorted(U, B \cdot X)$
→ $inserted(A, X, Y)$ & $B \leq A$ & $sorted(U, B \cdot X)$
$inserted(A, B \cdot X, A \cdot B \cdot X)$ & $sorted(U, B \cdot X)$
→ $A \leq B$ & $sorted(U, B \cdot X)$
$inserted(A, nil, A \cdot nil)$ → $true$

along with "more primitive" rules for inequality.

The specification for the above program may be stated as a conjunction of the requirements that the list *Z* be ordered in nondecreasing order and that it be a permutation of the list *X:*

ordered(Z) & permuted(X,Z) --> sorted (X ,Z).(1)

Given the facts

ordered (nil) -► true (2)

(an empty list is ordered) and

permuted (X ,X) --> true (3)

(any list is a permutation of itself), the desired base case

sorted (nil,nil) -> .true (4)

is generated.

To generate the recursive case, an auxiliary definition is required. To indicate the desire to sort a list by first sorting its tail, the following definition is added:

$$sorted(X, Y) \supset sorted(A \cdot X, Z) \rightarrow inserted(A, Y, Z). \quad (5)$$

From this, completion generates the following sequence of rules: Using the definition of ⊃ [i.e. $U \supset V \rightarrow (U \,\&\, V) = U$] to rewrite (5), gives

$$[sorted(A \cdot X, Z) \,\&\, sorted(X, Y)] = sorted(X, Y) \rightarrow inserted(A, Y, Z). \quad (6)$$

Using the associativity and commutativity of = (meaning equivalence) in overlapping the fact *U=U --> false* with (6), adding *—sorted(X,Y)* to both sides of the rules, gives

$$sorted(A \cdot X, Z) \,\&\, sorted(X, Y) \rightarrow inserted(A, Y, Z) = sorted(X, Y), \quad (7)$$

after simplifying the resultant left side with *U=false --> U.* Overlapping the fact *V &U-+U* with (7) adds the conjunct *sorted (X ,Y)* to both sides. Then simplifying the resulting critical pair, using propositional rules, gives

$$sorted(A \cdot X, Z) \,\&\, sorted(X, Y) \rightarrow inserted(A, Y, Z) \,\&\, sorted(X, Y). \quad (8)$$

The latter is the desired recursive call.

The next stage is to synthesize the auxiliary program. Adding the fact

permuted (A X ,AY) --> permuted (X X) (9)

will generate the two base cases

$$inserted(A, nil, A \cdot nil) \rightarrow true \quad (10)$$

$$inserted(A, B \cdot X, A \cdot B \cdot X) \,\&\, sorted(U, B \cdot X) \rightarrow A \leq B \,\&\, sorted(U, B \cdot X). \quad (11)$$

Generating the recursive case of *inserted* requires much more information about *ordered.* The predicate *ordered* can be defined in terms of *smaller:*

$$ordered(Y) \,\&\, smaller(B, Y) \rightarrow ordered(B \cdot Y), \quad (12)$$

i.e. a list beginning with an element *B* is ordered if (and only if) *B* is smaller than each element in the remainder of the list and the remainder is itself ordered. The predicate *smaller* can be defined by

$$smaller(B, nil) \rightarrow true \quad (13)$$

$$smaller(B, A \cdot X) \rightarrow B \leq A \,\&\, smaller(B, X) \quad (14)$$

In addition, one needs to know that

$$[smaller(B, X) \,\&\, permuted(Y, X)] \supset smaller(B, Y) \rightarrow true. \quad (15)$$

This fact must either be given as true, or can itself be proved using the completion procedure. (For the use of completion for inductive theorem proving, see, for example, [Huet-Oppen-80].) The result is

$$inserted(A, B \cdot X, B \cdot Z) \,\&\, B \leq A \,\&\, sorted(U, B \cdot X) \rightarrow inserted(A, X, Y) \,\&\, B \leq A \,\&\, sorted(U, B \cdot X). \quad (16)$$

7. IMPLEMENTATION

An implementation of these ideas is underway at the University of Illinois. It is being embedded within the rewrite system environments REVE [Lescanne-83] and RRL [Kapur-Sivakumar-83]. In practice, we have not encountered any difficulty in using the well-founded orderings supplied with these systems to successfully guide the synthesis of programs. Pruning futile paths, on the other hand, is a difficult problem.

ACKNOWLEDGEMENT

1 thank Alan Josephson for his critical readings and critical implementation work.

REFERENCES

[Boyer-Moore-79] Boyer, R. S., and Moore, J S. *A Commutational Logic.* Academic Press, New York, 1979.

[Burstall-Darlington-77] Burstall, R. M., and Darlington, J. "A transformation system for developing recursive programs". *J. of the Association for Computing Machinery,* Vol. 24, No. 1 (Jan. 1977), pp. 44-67.

[Burstall-MacQueen-Sannella-80] Burstall, R. M., MacQueen, D. B., and Sannella, D. T. "HOPE: An experimental applicative language". *Conference Record of the 1980 LISP Conference,* Stanford, CA (1980), pp. 136-143.

[Clark-81] Clark, K. L. "The synthesis and verification of logic programs", Research Report DOC 81/36, Department of Computing, Imperial College, London, England, Sept. 1981.

[Dershowitz-82] Dershowitz, N. "Orderings for term-rewriting systems". *J. Theoretical Computer Science,* Vol. 17, No. 3 (Mar. 1982), pp. 279-301.

[Dershowitz-85] Dershowitz, N. "Computing with rewrite systems". *Information and Control* (1985, to appear).

[Fages-83] Fages, F. "Formes canoniques dans les algebras booleennes, et application a la demonstration automatique en logique de premier ordre", These, Universite' de Paris VI, Paris, France, June 1983.

[Futatsugi,etal.-84] Futatsugi, K., Goguen, J. A., Jouannaud, J. P., and Meseguer, J. "Principles of OBJ2", Centre de Recherche en Informatique de Nancy, Nancy, France, 1984.

[Hoffmann-O'Donnell-82] Hoffmann, C. M., and O'Donnell, M. J. "Programming with equations". *Transactions on Programming Languages and Systems,* Vol. 4, No. 1 (Jan. 1982), pp. 83-112.

[Hogger-81] Hogger, C. J. "Derivation of logic programs". *J. of the Association for Computing Machinery,* Vol. 28, No. 2 (Apr. 1981), pp. 372-392.

[Hsiang-Dershowitz-83] Hsiang, J., and Dershowitz, N. "Rewrite methods for clausal and non-clausal theorem proving". *Proc. Tenth EATCS International Colloquium on Automata, Languages and Programming,* Barcelona, Spain (July 1983), pp. 331-346.

[Huet-Oppen-80] Huet, G, and Oppen, D. C. "Equations and rewrite rules: A survey". In: *Formal Language Theory: Perspectives and Open Problems,*

R. Book, ed. Academic Press, New York, 1980, pp. 349-405.

[Hullot-80] Hullot, J. M. "Compilation de formes canoniques dans les the'ories equationnelles", These, Universite' de Paris-Sud, Orsay, France, Nov. 1980.

[Kapur-Sivakumar-83] Kapur, D., and Sivakumar, G. "Experiments with and architecture of RRL, a rewrite rule laboratory". *Proc. NSF Workshop on the Rewrite Rule Laboratory,* Schenectady, NY (Sept. 1983), pp. 33-56.

[Knuth-Bendix-70] Knuth, D. E., and Bendix, P. B. "Simple word problems in universal algebras". In: *Computational Problems in Abstract Algebra,* J. Leech, ed. Pergamon Press, 1970, pp. 263-297.

[Kounalis-Zhang-85] Kounalis, E., and Zhang, H. "A general completeness test for equational specifications", Unpublished report, Centre de Recherche en Informatique de Nancy, Nancy, France, 1985.

[Kowalski-74] Kowalski, R. A. "Predicate logic as programming language". *Proc. IFIP Congress,* Amsterdam, The Netherlands (1974), pp. 569-574.

[Kowalski-79] Kowalski, R. A. *Logic for Problem Solving.* North-Holland, Amsterdam, 1979.

[Lescanne-83] Lescanne, P. "Computer experiments with the REVE term rewriting system generator". *Proc. Tenth Symposium on Principles of Programming Languages,* Austin, TX (Jan. 1983), pp. 99-108.

[Manna-Waldinger-80] Manna, Z., and Waldinger, R. J. "A deductive approach to program synthesis". *ACM Transactions on Programming Languages and Systems,* Vol. 2, No. 1 (Jan. 1980), pp. 90-121.

[Peterson-Stickel-81] Peterson, G. E., and Stickel, M. E. "Complete sets of reductions for some equational theories". *J. of the Association for Computing Machinery,* Vol. 28, No. 2 (Apr. 1981), pp. 233-264.

[Thiel-84] Thiel, J. J. "Stop losing sleep over incomplete data type specifications". *Proc. Eleventh Symposium on Principles of Programming Languages,* Salt Lake City, UT (Jan. 1984).

[Turner-79] Turner, D. A. "SASL language manual", University of St. Andrews, 1979.