J. Symbolic Computation (1985) **1**, 119–142

Automatic Programming: A Tutorial on Formal Methodologies

ALAN W. BIERMANN

Department of Computer Science,
Duke University, Durham, North Carolina 27706, *U.S.A.*

(*Received* 4 *March* 1985)

Ten methodologies for automatic program construction are presented, discussed and compared. Some of the techniques generate code from formal input–output specifications while others work from examples of the target behaviour or from natural language input.

1. Introduction

Computer programming is the process of translating a variety of vague and fragmentary pieces of information about a task into an efficient machine executable program for doing that task. *Automatic computer programming* or *automatic programming* occurs whenever a machine aids in this process.

The amount of automatic programming that is occurring is a variable quantity that depends on how much aid the human is given. There are a number of dimensions on which the level of help can be measured including the level of the language used by the human, the amount of informality allowed, the degree to which the system is told *what* to do rather than *how* to do it, and the efficiency of the resulting code. Thus we usually say that there is a higher degree of automatic programming whenever a higher level language is used, less precision is required of the human, the input instructions are more declarative and less procedural, and the quality of the object code is better.

The technologies of automatic programming thus include the fields that help move the programming experience along any of these dimensions: algorithm synthesis, programming language research, compiler theory, human factors, and others. This paper will concentrate on only the first of these topics, formal methodologies for the automatic construction of algorithms from fragmentary information.

The formal methodologies† have been separated into two categories, synthesis from formal specifications and synthesis from examples. In the former case, it is assumed a specification is given for the target program with adequate domain information so that the target program can be derived in a series of logical steps. In the later case, behavioural

This material is based on work supported by the Air Force Office of Scientific Research, Air Force Systems Command, USAF under Grant 81-0221 and the U.S. Army Research Office under Grant DAAG-29-84-K-0072.

† The formal methodologies that transform specifications, domain information and behavioural examples into programs operate on symbolic objects (logical formulae and programs) and aim at total or partial automatisation of this transformation process. Also, most of these methodologies are intimately connected with automatic theorem proving. Hence, automatic programming should be viewed in the context of symbolic computation as defined in the editorial of this journal.

0747–7171/85/020119+24 $03.00/0

examples are given for the desired program, and it is inferred by a series of generalisation steps.

After completing the coverage of these formal methodologies, a short section mentions some work on the generation of programs from natural language input using artificial intelligence knowledge based systems.

The various synthesis methodologies will be described by illustrating their operation on a single programming problem. In all cases, many details have been omitted or modified from the original sources to maintain brevity and readability. The reader is always encouraged to return to the referenced papers for more complete coverage.

2. Information Sources

The target program in a programming environment must be derived from basic information about the desired behaviour. Some of this information is furnished by the human user and some may be general domain knowledge. In all cases, enough information must be available to specify the target program.

However, the "structural distance" between the source information and the target may vary greatly. In some cases, the form of the axiomatic information may be quite close to that of the generated code, in which case we say that little automatic programming has occurred. In the extreme case, the user gives code in a compiled language that can be directly translated into the target language. In a less severe case, the user needs to specify axioms in a relatively exacting form, and the synthesiser performs only modest translations to assemble the target. In the most impressive cases, the source of information may be extremely random and fragmentary in nature leaving the synthesiser to discover essentially all of the structure required to do the computation.

For the sake of concreteness, we will study synthesis methodologies as they construct a program to remove the nonatomic entries from a list. Thus the desired behaviour transforms the input list $(A\ (B)\ C)$ to $(A\ C)$. If the input list has length zero, the output list is this same list, i.e. nil yields nil. The process of generating this program illustrates the construction of both a loop and a nested branch.

Relatively standard notations will be used in this paper If $x = (A\ (B)\ C)$ then *car* (x) will be a function that finds the first entry on the list, A. The function *cdr* (x) will return list x with the first entry removed, $((B)\ C)$ in this example. The *cons* (u, v) operator will add its first argument u to the front of its second argument, list v. Thus *cons* (x, x) will yield $((A\ (B)\ C)\ A\ (B)\ C)$. The symbol *atom* is a predicate which yields true if its argument is an atomic unit or list of length zero. Thus *atom* $((A\ (B)\ C))$ is FALSE and *atom* (A) is TRUE. The function *length* will give the length of a list. Thus *length* $((A\ (B)\ C))$ is 3.

In specifying the example problem given above, several key facts need to be included. Thus it will be necessary to give the fact that input nil is to yield output nil. This will be known as fact F1 in later discussions. It may be written as

$$\text{if } x = \text{nil then } f(x) = \text{nil} \qquad \text{(F1)}$$

where f is the name of the target program. But some synthesis methodologies will require other notations. For example, the notation $R(x, z)$ will denote the target relationship between the input x and output z. So the nil case is specified as $R(\text{nil}, \text{nil})$ or as $R(x, \text{nil}) \Rightarrow \text{TRUE}$ if $x = \text{nil}$. The $\Rightarrow$ symbol can be read as "may be replaced by".

If the input list x is not nil, its associated output z must also be specified. Considering the output list as a set, one can write $f(x) = z$ where

for all u[member(u, z) $\langle\!-\!\rangle$ (member(u, x) and atom (u))].

Another way of encoding this information is to say that

If not (x = nil) and not (atom (car(x))) and R(cdr(x), u)

then $R(x, u) \Rightarrow$ TRUE (F2)

and

if not (x = nil) and atom (car(x)) and R(cdr(x), u)

then $R(x, \text{cons}(\text{car}(x), u)) \Rightarrow$ TRUE (F3)

All of these forms will be used in the examples below.

The careful reader may object to synthesising programs from information as specific as the axioms given above. They are in some sense already executable in their current form and are thus programs as they stand. For example, the latter forms are almost identical to the following PROLOG program.

```
R(nil, nil) <-
R(x, y) <- R(cdr(x), y), not(atom(car(x))),
        not(atom(x))
R(x, cons(car(x), u)) <- R(cdr(x), u),
        atom(car(x)), not(atom(x))
```

Thus one can compute $f(x) = z$ by proving the theorem $R(x, z)$ on a PROLOG system. However, the synthesised programs given below are deterministic and thus compute much more efficiently than one can expect to execute specifications. The theorems implicit to the programs are already proved and there is no uncertainty related to their execution. The interpretation of specifications is dependent on the nature of the interpreter, the ordering of the axions, and other extraneous factors. The fact that specifications can be executed does not necessarily obviate the need for generating deterministic code.

3. Synthesis from Formal Specifications

Many methodologies have been developed in recent years for the generation of programs from specifications. Five of them will be described here: the *strategical approach* of Bibel & Hörnig (1984) which has been embedded in the LOPS system, the *divide and conquer* methodology of Smith (1985), the *transformational* technique as developed by Broy (1983), the *deductive sequent* method of Manna & Waldinger (1980), and the synthesis from *equational specifications* as described by Prywes *et al.* (1979).

3.1. A STRATEGICAL APPROACH

Bibel & Hörnig (1984) have constructed the logical program synthesis system LOPS which has a variety of facilities for acquiring specification information, manipulating domain knowledge, proposing critical theorems and proving them, and constructing code. We will follow their methodology as it solves the synthesis problem posed above. The system searches for relationships between the input and output and attempts to guess a portion of the input which can be selected out to begin computation of the output. If this is successful, it then attempts to find a recurrence of the original specification in the reduced form of the problem with part of the input already processed. If the recurrence

can be found, a loop is constructed to process the rest of the input and complete the synthesis.

Occasionally, the synthesis may be blocked because needed theorems are not available. Then the methodology invokes a model exploration capability to generate examples related to the unknown phenomena, to seek a generalisation of the examples observed, and to prove the generalisation if possible. Thus the approach attempts to include a capability similar to that of humans to find restatements or new relationships in the domain that can be key steps in the synthesis.

The methods begin with a dialogue with the user.

INITIALISE PROBLEM
f
INPUT VARIABLE
x
INPUT CONDITION
list(x)
OUTPUT CONDITION
if x = nil then $f(x)$ = nil
otherwise $f(x) = y$ where
for all u[member $(u, y) \langle - \rangle$ (member(u, x) and atom(u)]

Since the nil case is trivial, it can be handled immediately and all concerns can be transferred to the more general case.

$f(x)$ = if x = nil then nil else $g(x)$

The function $g(x)$ needs to be built to process all other inputs.

$g(x) = y$ where not(x = nil) and list (y) and
for all u[member(u, y) $\langle - \rangle$ (member(u, x) and atom(u))]

At this point, the processor attempts to decompose the problem in such a way that part of the calculation can be done immediately and the rest can be done later by a recursive call to g. Bibel & Hörnig argue that there are relatively few practical recursion schemes and that their system needs only to examine those few.

To initiate this loop finding behaviour, the system "guesses" what the output might be and which piece of the input should be processed first. In the current example, we assume it selects the first entry in x. Since the only processing done in this example problem is the transfer of items from input to output, the only question is whether this first entry is in or not in the output.

$g(x) = y$ where not(x = nil) and list(y) and
for all u[member(u, y) $\langle - \rangle$ (member(u, x) and
atom(u)] and
[not(member(car(x), y)) eor member (car(x), y)]

Here eor means "exclusive or". This can be rewritten to consider the two cases separately, not (member(car(x), y)) and member(car(x), y).

$g(x) = y$ where not(x = nil) and list(y) and
[(for all u[member(u, y) $\langle - \rangle$ (member(u, x) and atom(u))]
and (not(member(car(x), y))))
eor
(for all u[member(u, y) $\langle - \rangle$ (member(u, x) and atom(u))]
and member(car(x), y))]

Next, the theorem prover invokes the fact that if car(x) is not a member of the output, then one can construct y without using it, i.e. use cdr(x) instead of x. If car(x) is in y, then it must be added to the front of y.

$g(x) = y$ where not(x = nil) and list(y) and
[(for all u[member(u, y) <–> (member(u, cdr(x)) and atom(u))]
and (not(member(car(x), y))))
eor
(there exists y'
(for all u[member(u, y') <–> (member(u, cdr(x)) and atom(u))]
and y = cons(car(x), y')
and member(car(x), y)))]

Loop finding has, in fact, succeeded at this point because copies of the original specification for f can be found. Thus g can now be written in terms of the function f.

$g(x) = y$ where not (x = nil) and list(y) and
[($y = f$(cdr(x)) and not (member(car(x), y)))
eor
($y' = f$(cdr(x))and y = cons(car(x), y')
and member(car(x), y))]

(At each step, of course, many details have been omitted.) This can be rewritten as

$g(x) = y$ where not(x = nil) and list(y) and
[($y = f$(cdr(x)) and not(member(car(x), y)))
eor
(y = cons(car(x), f(cdr(x)))
and member(car(x), y))]

Finally, the system needs to be able to find property p such that

p(car(x)) <–> member(car(x), y).

It is possible that the knowledge base and theorem prover will be adequate to discover that p is the predicate atom. If the theorem prover cannot discover this fact, the model generator is invoked to generate typical cases. Then a generalisation is attempted to hypothesise what key property controls membership in y. Then if the theorem prover confirms the hypothesis, the predicate p can be used in the program. The final version of g is

$g(x)$ = if atom (car(x)) then cons(car(x), f(cdr(x)))
else f(cdr(x))

Of course, from earlier considerations it was decided that

$f(x)$ = if x = nil then nil else $g(x)$.

An important point about this approach is that it attempts to avoid dependence on axioms that already specify the target loop through their embedded recursions as appear in F2 and F3. The methodology depends on theorem proving methodologies and the strategy of generalising from examples to obtain this level of performance.

3.2. DIVIDE AND CONQUER

Software engineers have often advocated that programs be constructed in a "top down" manner. The original specifications for a program are broken into parts which

may be simple enough to solve directly or which may become non-trivial subtasks to be solved individually. The subtasks may be similarly decomposed into even smaller tasks and so forth until all parts are primitive enough for solution. Then the individual parts are recomposed beginning at the bottom of the tree until a complete solution is assembled.

The *divide and conquer* methodology of Smith (1985) follows this strategy by partitioning the problem on the dimension of the data being processed. That is, if f is to be constructed which has input x, the methodology first checks to see whether the result can be directly computed. If not it divides x into parts and operates on the parts of x individually. Then the results of the individual calculations are assembled to produce the result of f. Symbolically, this idea can be expressed as follows:

$f(x)$ = if primitive(x) then directly solve (x)
else compose. ($f_1 \times f_2$). decompose(x)

Here the result of decompose is a vector of length two and the periods indicate function composition. The ith entry of the decomposition is operated on by f_i for each $i = 1, 2$ resulting in another such vector. Then compose assembles the parts of the vector to yield the output for f.

This methodology is interesting because of its structural simplicity, because it often leads to efficient programs, and because of its ubiquity in practical situations. We will examine its application in solving the problem posed in the previous section.

The synthesis begins with the primitive case which appears in the specifications as F1.

$f(x)$ = if x = nil then nil

Then the method of decomposition must be selected. Many choices may be suggested, and in practice the synthesis process may be forced to try them all. For brevity here, we will examine a choice that leads to a successful synthesis, decompose = (car × cdr). That is, the input list will be broken into two parts, the first item and the rest of the list, and the computation will be dependent on processing them separately and assembling the solution.

In the case where x is not nil, our definition states

$f(x) = z$ where
for all u[member(u, z) $\langle\text{–}\rangle$ (member(u, x) and atom(u))].

The above decomposition suggests a partition on the values of u.

$f(x) = z$ where
for u = car(x) and for all u in cdr(x)
[member(u, z) $\langle\text{–}\rangle$ (member(u, x) and atom(u))]

Thus the output z has two parts, that part coming from u = car(x) and that from u in cdr(x).

$f(x) = z$ where
z = append (
z_1 such that u = car(x)[member(u, z_1) $\langle\text{–}\rangle$ (member(u, x) and atom(u))]
z_2 such that $u \in$ cdr(x)[member(u, z_2) $\langle\text{–}\rangle$ (member(u, x) and atom(u))])

(The function append joins two lists to form a single list.)

Minor simplifications lead to the following:

$f(x)$ = append(

z_1 such that [member(car(x), z_1) $\langle - \rangle$ atom(car(x))],
z_2 such that for all u
[member(u, z_2) $\langle - \rangle$ (member(u, cdr(x)) and atom (u))])

This leads to the definition

$$z_1 = f_1\ (\text{car}(x)) = \text{if atom}(\text{car}(x)) \text{ then cons}(\text{car}(x), \text{nil}) \text{ else nil}$$

since z_1 is the list containing car(x) if atom(car(x)) and nil otherwise.

We also note that z_2 is defined in terms of the original definition of f.

$$z_2 = f(\text{cdr}(x))$$

Thus we conclude for the case not(x = nil) that

$$\begin{aligned} f(x) &= \text{append}(f_1(\text{car}(x)), f(\text{cdr}(x))) \\ &= \text{append} \,.\, (f_1 \times f) \,.\, (\text{car} \times \text{cdr})x \end{aligned}$$

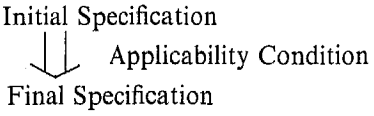
The final synthesised program is

$$\begin{aligned} f(x) = &\text{ if } x = \text{nil then nil} \\ &\text{else append} \,.\, (f_1 \times f) \,.\, (\text{car} \times \text{cdr})x \end{aligned}$$

Smith (1985) has shown how this strategy can be used to create a variety of programs. An obvious domain for study is the class of sorting programs, many of which can be derived by making different choices for the decomposition operator. Thus a car-cdr choice as used above leads to an insertion sort. If the decomposition is a split, a merge sort results and other choices lead to such algorithms as selection sort, quicksort, and others.

3.3. TRANSFORMATIONAL METHODOLOGIES

A popular approach to program synthesis in recent years (Broy, 1983; Burstall & Darlington, 1977; Gerhart, 1976; Manna & Waldinger, 1979) follows the technique of sequentially transforming the original specification until a program is derived. Such transformations take the form



where the *initial specification* is a schema giving the form of the specification to be modified, the *applicability condition* specifies all relationships that are prerequisite to use of the transformation, and the *final specification* gives the revised form of the initial specification. Program synthesis begins with the original specification and applies rules of this form until the target program is complete.

The methodology assumes the availability of a variety of transformations and the program derivation involves selecting the appropriate ones and instantiating them to allow convergence to a satisfactory program. Early transformations in a given construction may have the effect of modifying the original specifications into more convenient form. Later transformations begin to assemble parts of the program. The final transformations may manipulate a nearly complete code segment to increase efficiency.

A feeling for the general approach can be obtained by rereading one of the previous sections and constructing formal transformations to achieve the sequential steps which are less well justified in their current form. Transformations can be designed for splitting a

problem into cases, for collapsing disparate cases into a single set, for creating looping and branching code, subroutines, and many other constructions. The above references give numerous examples of such transformations.

A very useful characteristic of the transformational approach is that it allows the embedding of arbitrary amounts of information into a single step. Transformations may be small and simple to build one construct at a time. They can also be large enough to do a huge fraction of the work with one application.

An example of the latter point is the following transformation which stores some of the insights of the above derivation in a single structure.

$$f(x) = z \text{ where } R(x, z)$$

$$\Big\downarrow\Big\downarrow \quad \text{Applicability Conditions}$$

$$f(x) = \text{if } C(x) \text{ then } T(x) \text{ else } G(H_1(x), f(H_2(x)))$$

where the applicability conditions are

$$C(x) \rightarrow R(x, T(x))$$

$$\neg C(x) \text{ and } R(H_2(x), y) \rightarrow R(x, G(H_1(x), y))$$

$$\text{length}(x) = 0 \rightarrow C(x)$$

$$\text{length}(x) > 0 \rightarrow \text{length}(H_2(x)) < \text{length}(x)$$

This transformation is applicable to the example of this paper. The only problem is to find instantiations of the notations so that the applicability conditions will hold. From the considerations of the previous section, it is clear that they are all true in the following case:

$C(x)$ is $x = \text{nil}$
$T(x)$ is nil
$H_1(x) = f_1\ (\text{car}(x))$
$H_2(x) = \text{cdr}\ (x)$
$G(x, y) = \text{append}\ (x, y)$

3.4. THE DEDUCTIVE SEQUENT

Manna & Waldinger (1980) have developed a tabular method for organizing the derivation of programs using a deductive technique. The methodology is built around the concept of a *sequent* which is a three column table with rows of the form

Assertions	*Goals*	*Outputs*
$A_i(a, x)$		$t_i(a, x)$

or

Assertions	*Goals*	*Outputs*
	$G_j(a, x)$	$t_j(a, x)$

This table has the meaning

if for all x, $A_1(a, x)$ and
for all x, $A_2(a, x)$ and
—
—
—
for all x, $A_m(a, x)$

then

for some x, $G_1(a, x)$ or
for some x, $G_2(a, x)$ or
—
—
—
for some x, $G_n(a, x)$

where a denotes the constants and x denotes the free variables in the domain. The output entries in the sequent have the following meaning: If some instance of a goal is true, then its corresponding output satisfies the specification. If some instance of an assertion is false, its corresponding output satisfies the specification.

Program synthesis proceeds as follows. The original specification is properly coded into the sequent and then a series of new rows are added in an attempt to deduce the target program. If a goal is reached that is true or an assertion is found that is false with the associated output entry in terms of computational primitives, that output entry is a program satisfying the original specification.

The initial specification is

for all a, for some $z(P(a) \rightarrow R(a, z))$

where $P(a)$ is a specification on the input and R is as defined in earlier sections. In the example problem of this paper, $P(a)$ may be taken to require that the input be a list. In the sequent, this appears as follows:

Assertions	*Goals*	*Outputs*	
$P(a)$			(1)
	$R(a, z)$	z	(2)

We seek a sequence of deductive steps that will achieve the target program.

Assertions	*Goals*	*Outputs*
	TRUE	(Program)

Manna & Waldinger have given several mechanisms for deducing new rows in the sequent. One utilises transformations of the form

$r \Rightarrow s$ if P.

Such a transformation can be used to replace a subexpression in an assertion or goal as follows. Suppose, for example, a goal F has subexpression r' such that there is a unifier θ for r and r', i.e. $r\theta = r'\theta$. Then goal F can be replaced by goal $P\theta$ and $F[r\theta \leftarrow s\theta]$. The notation $F[r\theta \leftarrow s\theta]$ means $r\theta$ in F is replaced by $s\theta$. If F has a corresponding output t, the new goal will have output $t\theta$. This appears in the sequent as shown.

Assertions	*Goals*	*Outputs*
	F	t
	$P\theta$ and $F\theta[r\theta \leftarrow s\theta]$	$t\theta$

As an illustration, the transformation F1 from Section 2 can be applied to goal (2) above. The unifier θ is $x \leftarrow a$ and $z \leftarrow$ nil. That is $R(a, \text{nil}) \Rightarrow$ true if $a = \text{nil}$ converts $R(a, z)$ to $(a = \text{nil})$ and true.

Assertions	*Goals*	*Outputs*	
	$(a = \text{nil})$	nil	(3)

Similarly, transformations F2 and F3 can also be applied to (2).

Assertions	*Goals*	*Outputs*	
	not(a = nil) and not(atom(car(a))) and R(cdr(a), u)	u	(4)
	not(a = nil) and atom(car(a)) and R(cdr(a), u)	cons(car(a), u)	(5)

Another way to add rows to the sequent is to resolve two goals to obtain a new goal. Suppose F and G are goals with output entries t_1 and t_2.

Assertions	*Goals*	*Outputs*
	F	t_1
	G	t_2

Further, suppose F and G have subsentences P_1 and P_2 which can be unified by θ: $P_1\theta = P_2\theta$. Then a new goal can be created $F\theta[P_1\theta \leftarrow \text{TRUE}]$ and $G\theta[P_2\theta \leftarrow \text{FALSE}]$ with output entry if $P_1\theta$ then $t_1\theta$ else $t_2\theta$.

Assertions	*Goals*	*Outputs*
	$F\theta[P_1\theta \leftarrow \text{TRUE}]$ and $G\theta[P_2\theta \leftarrow \text{FALSE}]$	if $P_1\theta$ then $t_1\theta$ else $t_2\theta$

This is called a *GG-resolution*. As an example of this deductive step, suppose goal(3) is resolved with the following goal.

Assertions	*Goals*	*Outputs*
	not(a = nil)	t

Then θ can be nil and $P_1\theta = P_2\theta = (a = \text{nil})$. The resolution produces this new row:

Assertions	*Goals*	*Outputs*
	TRUE and not FALSE	if a = nil and then nil else t

Looping is introduced in the sequent by adding an induction hypothesis as an assertion and then resolving it with other rows. For example, in (4) above we see the relation R has been deduced with a reduced input cdr(a). This suggests that a recursive loop can possibly be derived so the inductive hypothesis is added.

Assertions	*Goals*	*Outputs*	
if $u < a$ then if $P(u)$ then $R(u, f(u))$			(6)

That is, if u is less than input a by some well-founded ordering, we are assuming the program f realises the target relation R for that u. One can do a *GA resolution* similar to the GG resolution above to combine (4) and (6) to obtain a new goal.

Assertions	*Goals*	*Outputs*	
	not(a = nil) and not(atom(car(a)))	f(cdr(a))	(7)

Looking at the output, we see that a recursive call to f has, in fact, been synthesised.

Similarly, another GA resolution can be made between (5) and (6) to obtain another recursive form.

Assertions	*Goals*	*Outputs*	
	not(a = nil) and atom(car(a))	cons(car(a), f(cdr(a)))	(8)

The target program can be derived by combining (7) and (8) with a GG resolution to obtain (9) and similarly (3) and (9) to obtain (10).

Assertions	*Goals*	*Outputs*	
	not(a = nil)	if not(atom(car(a))) then f(cdr(a)) else cons(car(a), f(cdr(a)))	(9)
	TRUE	if a = nil then nil else if not (atom(car(a))) then f(cdr(a)) else cons(car(a), f(cdr(a)))	(10)

The Manna–Waldinger deductive system combines many of the program synthesis mechanisms developed during the 1970s into one unified and systematic approach. The ideas of constructing code as a side effect to theorem proving, generalised resolution, transformational techniques and others all are utilised in the method simultaneously and compatibly.

3.5. SYNTHESIS FROM EQUATIONAL SPECIFICATIONS

Prywes *et al.* (1979) have developed a methodology for program specification which has some resemblances to the PROLOG language. Declarative information is given by the user regarding the relationships between objects in the domain. The automatic system then executes the statements in an order that may be unrelated to their presentation order and in a chaining manner to find answers. Some of the differences with PROLOG are that the Prywes language MODEL is aimed at business applications and is especially designed to compile into efficient object code.

The MODEL language is not designed to do symbolic computations of the kind required by the example problem of this paper. However, we will modify its syntax slightly and invent enough new constructs to show how it might handle the problem. The basic data structure is the array and we will assume pairs of the form (i, x) can fit into individual array entries. Here i is an integer and x is the string being processed. Integer i indicates which item in x is to be processed next.

The pseudo-MODEL program utilises an array in which the sequential iterations in the computation are stored. The second equation specifies the inductive step which builds a looping behaviour to complete the computation.

$A(1) = (1, x)$
$A(I) =$ if atom(select(i, x) in $A(I\text{-}1)$) then $(i+1, x)$
else (i, delete (i, x) in $A(I\text{-}1)$)

j = LEAST(I, $A(I) = (\text{length}(v)+1, v)$)
RESULT = z such that $A(j) = (k, z)$

The function (select(i, x) in $A(I\text{-}1)$) finds (i, x) in $A(I\text{-}1)$ and returns the ith entry in list x. The function (delete(i, x) in $A(I\text{-}1)$) acts similarly except that it deletes the ith entry in list x.

Tracing the operation of this program on the sample input $x = (A\ (B)\ C)$ yields

$A(1) = (1, (A\ (B)\ C))$
$A(2) = (2, (A\ (B)\ C))$
$A(3) = (2, (A\ C))$
$A(4) = (3, (A\ C))$
$j = 4$
$z = (A\ C)$

Strategies for compiling this language will not be discussed here but are described in Pnueli *et al.* (1984).

4. Synthesis from Examples

Suppose $p_1, p_2, p_3, \ldots$, is an enumeration of all programs in the space of possible programs and that p_i is the first program that meets the user's needs. Then a satisfactory specification for p_i is enough behavioural information to separate it from all of its predecessors. One synthesis procedure thus is to simply enumerate programs checking each one for acceptability until p_i is found.

This approach to synthesis is desirable from the user's point of view because experience shows that relatively little behavioural information is typically needed to disqualify the predecessors of p_i from consideration. However, from the point of view of the system designer, it is fraught with difficulties. First, the undecidability of the halting problem makes it impossible to reliably determine whether a given input–output behaviour can be achieved for each program. Second, the cost of enumerating and checking all predecessors of p_i is normally too high to be attempted.

Any approach to synthesis must deal with these problems. The halting problem can be managed by requiring some kind of execution time limitation be included with example behaviours. The cost of enumeration can sometimes be controlled by requiring that enough structural information be given about the computation of the examples so that most predecessors can be efficiently skipped allowing fast convergence to the answer. In some synthesis environments, the input information is sufficient to eliminate search. In most research on synthesis from examples, only very limited classes of programs are considered so that the effects of both problems are minimised.

This section will consider four approaches to program synthesis from examples: the *function merging technique* which can construct any *regular* LISP program from its examples, the synthesis of single loop LISP programs using recurrence relations, the generation of the class of LISP *scanning programs* using a production rule method, and the synthesis of programs by generalisation of logic characterisations.

4.1. THE FUNCTION MERGING TECHNIQUE

In the example problem the desired output from the input $x = (A\ (B)\ C)$ is $z = (A\ C)$. It is straightforward to write z in terms of x.

$$z = \text{cons}(\text{car}(x), \text{cons}(\text{car}(\text{cdr}(\text{cdr}(x))), \text{cdr}(\text{cdr}(\text{cdr}(x)))))$$

That is, z is built by "consing" together three atoms found in x: A, C, and the nil obtained by three cdr operations on x. This decomposition is a major step toward building a program to do the target computation. The synthesis proceeds by assembling this set of primitives into a compact program.

The second step in the synthesis involves breaking the functions used to compute z into a series of primitive steps of the form

$$\begin{aligned} f_i(x) &= \text{cons}(f_j(x), f_k(x)) \\ f_i(x) &= f_j(\text{car}(x)) \\ f_i(x) &= f_j(\text{cdr}(x)) \\ f_i(x) &= x \end{aligned}$$

In fact, z decomposes as follows:

$$\begin{aligned} z = f_1(x) &= \text{cons}(f_2(x), f_3(x)) \\ f_2(x) &= f_4(\text{car}(x)) \\ f_3(x) &= \text{cons}(f_5(x), f_6(x)) \\ f_4(x) &= x \\ f_5(x) &= f_7(\text{cdr}(x)) \\ f_6(x) &= f_8(\text{cdr}(x)) \\ f_7(x) &= f_9(\text{cdr}(x)) \\ f_8(x) &= f_{10}(\text{cdr}(x)) \\ f_9(x) &= f_{11}(\text{car}(x)) \\ f_{10}(x) &= f_{12}(\text{cdr}(x)) \\ f_{11}(x) &= x \\ f_{12}(x) &= x \end{aligned}$$

It turns out that this is an inefficient computation of z. For example, we notice that

$$\begin{aligned} f_3(x) &= \text{cons}(f_5(x), f_6(x)) \\ f_5(x) &= f_7(\text{cdr}(x)) \\ f_6(x) &= f_8(\text{cdr}(x)) \end{aligned}$$

If only $\text{cdr}(x)$ is to be used in computing $f_3(x)$, one should compute the value of $\text{cdr}(x)$ before passing control to cons. Thus the computation can be done with just two primitive functions.

$$\begin{aligned} f_3(x) &= f_5(\text{cdr}(x)) \\ f_5(x) &= \text{cons}(f_7(x), f_8(x)) \end{aligned}$$

Thus cdr (and car) operations can be pushed above a cons operation in some cases to increase efficiency. If all possible such improvements are made, the functional decomposition is shortened maximally. (For reasons of the later discussion, the input of each function is also listed.)

$$\begin{aligned} z = f_1(x) &= \text{cons}(f_2(x), f_3(x)) && x = (A\ (B)\ C) \\ f_2(x) &= f_4(\text{car}(x)) && x = (A\ (B)\ C) \\ f_3(x) &= f_5(\text{cdr}(x)) && x = (A\ (B)\ C) \\ f_4(x) &= x && x = A \\ f_5(x) &= f_6(\text{cdr}(x)) && x = ((B)\ C) \\ f_6(x) &= \text{cons}(f_9(x), f_{10}(x)) && x = (C) \end{aligned}$$

$$
\begin{aligned}
f_9(x) &= f_{11}(\text{car}(x)) && x = (C)\\
f_{10}(x) &= f_{12}(\text{cdr}(x)) && x = (C)\\
f_{11}(x) &= x && x = C\\
f_{12}(x) &= x && x = \text{nil}
\end{aligned}
$$

The function merging technique finishes its task by finding the maximal merge of these functions consistent with the goal of doing the sample computation. There are two strategies for merging functions. First, if two functions have identical form they merge trivially.

$$f_4(x) = f_{11}(x) = f_{12}(x) = x$$

If two functions $f_i(x)$ and $f_j(x)$ have different form, it still may be possible to merge them by using the LISP conditional cond. For example, we notice that f_4, f_{11}, and f_{12} are all called with arguments that are atoms. So they could be merged with a function like f_1 which is not called with an atomic argument.

$$
\begin{aligned}
f(x) = \text{cond } &(\text{atom}(x)\ x)\\
&(T\ \text{cons}(f_2(x), f_3(x)))
\end{aligned}
$$

This function thus checks the argument x and returns the value that f_4, f_{11}, or f_{12} would give if x is an atom. Otherwise it returns $\text{cons}(f_2(x), f_3(x))$. Thus it can substitute for these three functions and f_1 as well. Merges of this kind are possible if predicates can be found to enable a branch down the correct path wherever necessary to duplicate the computation of the unmerged sequence of functions. The function merging approach usually employes only a limited class of predicates so the construction of the necessary tests for a given merge is inexpensive.

This merging can thus be continued if we note that predicates can be found to differentiate the argument of f_4, f_{11}, f_{12} (atom(x)) from the argument of f_1 (atom(car(x))) which is different from the argument of f_5(other). This merger, in fact, leaves f as a three-way branch.

$$
\begin{aligned}
f(x) = \text{cond } &(\text{atom}(x)\ x)\\
&(\text{atom } (\text{car}(x))\ \text{cons}(f_2(x), f_3(x)))\\
&(T f_6(\text{cdr}(x)))
\end{aligned}
$$

The maximal merger consistent with the execution of the original example partitions the functions into three groups,

$$\{f_1, f_4, f_5, f_6, f_{11}, f_{12}\},\ \{f_2, f_9\},\ \text{and}\ \{f_3, f_{10}\}.$$

The resultant program is

$$
\begin{aligned}
f(x) = \text{cond } &(\text{atom}(x)\ x)\\
&(\text{atom}(\text{car}(x))\ \text{cons}(f_2(x), f_3(x)))\\
&(T\ f(\text{cdr}(x)))\\
f_2(x) &= f(\text{car}(x))\\
f_3(x) &= f(\text{cdr}(x))
\end{aligned}
$$

In summary, the function merging technique has four major steps.

1. Write z in terms of x using cons, car, and cdr functions.
2. Break this computation into a set of primitive functions f_i of the form described above.

3. Revise this computation trace for maximum efficiency by lifting car and cdr operations above cons operations.

4. Find a maximal merge of the f_i functions by introducing cond branches wherever possible.

This synthesis procedure was developed by Biermann (1978), who has shown it can create any program in the class of *regular* LISP functions on the basis of randomly selected examples. The regular LISP programs are, roughly speaking, all those programs which can be constructed from the primitives described above, which use no auxiliary variables, and which use predicates made up of atom operating on a nesting of car and cdr functions. This synthesis procedure is functionally equivalent to a full enumeration, and therefore has the desirable property of guaranteed convergence to a solution if one exists in the class of regular LISP programs. It also has the disadvantage of being exponentially expensive on the size of the target program.

Historically, the function merging technique evolved from the *node merging technique* of Biermann (1972), where a "trainable Turing machine" was defined and studied. The trainable Turing machine had a learning mode in which the user could force the read-write head up and down the tape manually simulating the desired calculation. During this phase, the system stored a trace of the one or several calculations and used node merging to collapse those traces into a minimal finite state controller. Then the Turing machine could be switched to compute mode and use the controller it had created. For example, it was shown in Biermann *et al.* (1975) that this machine could be shown how to simulate one simple Turing machine and then program itself to be a universal Turing machine. Biermann & Krishnaswamy (1976) used the node merging technique in another application to build a self-programming desk calculator.

4.2. SYNTHESIS FROM RECURRENCE RELATIONS

The high cost of enumerative methods can be avoided if one increases the information in the examples and limits consideration to the class of single loop programs. First, we input a sequence of examples to illustrate the performance on sequentially larger inputs. Then recurrence relations are constructed which can be used to create the program.

For the standard example of this paper, we begin with the following input–output information.

Example	*Input*	*Output*
1	nil	nil
2	(*C*)	(*C*)
3	((*B*) *C*)	(*C*)
4	(*A* (*B*) *C*)	(*A C*)
5	((*D*) *A* (*B*) *C*)	(*A C*)

Then each output is written down in terms of its associated input using primitive functions.

$f_1(x) = \text{nil}$
$f_2(x) = \text{cons}(\text{car}(x), \text{nil})$
$f_3(x) = \text{cons}(\text{car}(\text{cdr}(x)), \text{nil})$
$f_4(x) = \text{cons}(\text{car}(x), \text{cons}(\text{car}(\text{cdr}(\text{cdr}(x))), \text{nil}))$
$f_5(x) = \text{cons}(\text{car}(\text{cdr}(x)), \text{cons}(\text{car}(\text{cdr}(\text{cdr}(\text{cdr}(x)))), \text{nil}))$

Studying these f_i's, one can find recurrences; for i greater than 1, each f_i can be written in terms of f_{i-1}.

Input	*Recurrence relation*
nil	$f_1(x) = \text{nil}$
(C)	$f_2(x) = \text{cons}(\text{car}(x), f_1(\text{cdr}(x)))$
$((B)\ C)$	$f_3(x) = f_2(\text{cdr}(x))$
$(A\ (B)\ C)$	$f_4(x) = \text{cons}(\text{car}(x), f_3(\text{cdr}(x)))$
$((D)\ A\ (B)\ C)$	$f_5(x) = f_4(\text{cdr}(x))$

From these relations it is apparent that the lowest level computation for $x = \text{nil}$ is $f_1(x) = \text{nil}$ and that two recurrences can be observed,

$$f_i(x) = \text{cons}(\text{car}(x), f_{i-1}(\text{cdr}(x)))$$

and

$$f_i(x) = f_{i-1}(\text{cdr}(x)).$$

Thus the program has the form

$f(x) =$ if atom(x) then nil
else (select correct recurrence).

The predicate generation can run on the inputs shown above and attempt to find a predicate that will differentiate the inputs associated with $\text{cons}(\text{car}(x), f_{i-1}(\text{cdr}(x)))$ from those associated with $f_{i-1}(\text{cdr}(x))$. In fact, atom(car(x)) is sufficient yielding the final program.

$f(x) =$ if atom(x) then nil else
if atom(car(x)) then cons(car(x), f(cdr(x)))
else f(cdr(x))

Summers (1977) has developed a theory of LISP synthesis from examples, and Kodratoff & Jouannaud (1984) have extended it. The methodology provides a format for discovering recurrence relations and a theorem that gives the exact form of the required program to achieve the behaviour specified by those relations. The proof of the theorem guarantees that if the recurrence relations characterise the target behaviour, the given program will correctly realise it.

4.3. A PRODUCTION RULE METHODOLOGY

Another strategy for generating programs is to diagnose their behaviours in a hierarchical manner and then generate the required code using production rules. Starting, for example, from the knowledge that $(A\ (B)\ C)$ is to produce $(A\ C)$, one can represent the required input output behaviour with the graph of Fig. 1. This methodology is particularly aimed at transferring items from the input to the output possibly in a sequence of scans and with modifications in order. Each local problem is diagnosed in a graph similar to Fig. 1 and a small set of production rules are available to solve the problems as they arise.

The approach begins by proposing code to solve the lowest level problem, the construction of the output list. There is an appropriate production rule and it is selected.

$[P_w^0, (X_0\ XL), \text{next}] \Rightarrow$
$P_w^0(X_0, XL) = \text{cons}(\text{car}(X_0), \text{next})$

This rule consists of two parts, a non-terminal symbol in square brackets and a line of generated code. The non-terminal symbol is quite complicated because it contains many parameters that will be instantiated when the rule is used. This non-terminal means that program P_w^0 is to be defined with arguments $(X_0\ XL)$ and the result of the computation is to be appended to "next".

This program P^0 needs to be called twice to obtain the output. This is represented in Fig. 2. Next, diagnosis observes that P^0 is to be invoked for two, but not all of the inputs, so that a conditional rule is needed. It builds a predicate to separate the two behaviours and then proposes the conditional rule.

$$
\begin{aligned}
&[P_w^1, (X_1\ XL), \text{next}] \Rightarrow \\
&P_w^1(X_1, XL) = \\
&\text{cond } (\text{atom}(\text{car}(X_1)) P_{w1}^0(X_1, X_1, XL)) \\
&\qquad\qquad (T\ \text{next}) \\
&[P_{w1}^0, (X_0, X_1, XL), \text{next}]
\end{aligned}
$$

This rule generates two things, some code to generate the branching behaviour and a non-terminal to generate the lower level construction of the output. The result of this diagnosis is shown in Fig. 3. Code is needed to call routine P^1 for each item on the input list.

Finally, code is needed to call P^1 three times as shown in Fig. 3. A production rule for generating looping code is selected.

$$
\begin{aligned}
&[P_w^2, (X_2\ XL), \text{next}] \Rightarrow \\
&P_w^2(X_2, XL) = \\
&\text{cond}(\text{atom}(X_2)\ \text{next}) \\
&\qquad\qquad (T\ P_{w1}^1(X_2, X_2, XL)) \\
&[P_{w1}^1, (X_1 X_2\ XL), P_w^2(\text{cdr}(X_2), XL)]
\end{aligned}
$$

This program will make all of the required calls in Fig. 3 and thus reduces the problem to that shown in Fig. 4. Program P^2 needs to be called with input $(A\ (B)\ C)$.

When the problem has been reduced to one call as shown in Fig. 4, it has been solved and the code can be generated by expanding the non-terminal $[P^2, (X_2), \text{nil}]$ with the three rules given above. Applying the P^2 rule with w and XL set to be null strings and next = nil, we obtain the first step in code generation.

$$
\begin{aligned}
&[P^2, (X_2), \text{nil}] \Rightarrow \\
&P^2(X_2) = \text{cond } (\text{atom}(X_2)\ \text{nil}) \\
&\qquad\qquad (T\ P_1^1(X_2, X_2)) \\
&[P_1^1, (X_1, X_2), P^2(\text{cdr}(X_2))]
\end{aligned}
$$

This step has produced another non-terminal which can be expanded using the branching rule with $w = 1$, $XL = X_2$, next $= P^2(\text{cdr}(X_2))$.

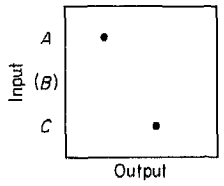


Fig. 1. Graphing the target behaviour.

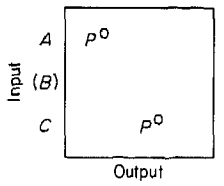


Fig. 2. Proposing P^0 to construct the output.

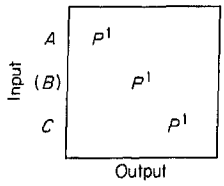


Fig. 3. Proposing branching code to separate the kinds of input.

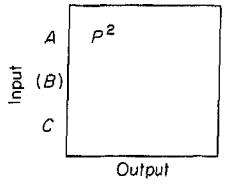


Fig. 4. Proposing looping code to call P^1.

$$[P_1^1, (X_1\ X_2), P^2(\text{cdr}(X_2))] \Rightarrow$$
$$P_1^1(X_1, X_2) =$$
$$\text{cond } (\text{atom}(\text{car}(X_1))P_{11}^0(X_1, X_1, X_2))$$
$$(T\ P^2(\text{cdr}(X_2)))$$
$$[P_{11}^0, (X_0, X_1, X_2), P^2(\text{cdr}(X_2))]$$

Now the P^0 rule is needed to remove the last generated non-terminal with $w = 11$, $XL = X_1X_2$, and next $= P^2(\text{cdr}(X_2))$.

$$[P_{11}^0, (X_0X_1X_2), P_2(\text{cdr}(X_2))] \Rightarrow$$
$$P_{11}^0(X_0, X_1, X_2) = \text{cons}(\text{car}(X_0), P^2(\text{cdr}(X_2)))$$

The union of the above generated code is the final program.

$$P^2(X_2) = \text{cond}(\text{atom}(X_2)\ \text{nil})$$
$$(T\ P_1^1(X_2, X_2))$$
$$P_1^1(X_1, X_2) = \text{cond}(\text{atom}(\text{car}(X_1))P_{11}^0(X_1, X_1, X_2))$$
$$(T\ P^2(\text{cdr}(X_2)))$$
$$P_{11}^0(X_0, X_1, X_2) = \text{cons}(\text{car}(X_0), P^2(\text{cdr}(X_2)))$$

Biermann & Smith (1979) have developed the production rule technique described here which generates *scanning* code, programs which repeatedly scan the input and transfer tokens to the output in various orders and forms. It is shown that a set of six rule schemas are enough to generate a large class of programs. This methodology is particularly unusual in its ability to diagnose and synthesise deeply nested loops.

4.4. SYNTHESIS THROUGH GENERALISATION ON LOGICAL ASSERTIONS

A number of researchers (Cohen & Sammut, 1984; Michalski, 1980; Shapiro, 1981) have examined techniques for inferring general statements about a phenomenon from specific facts. It turns out that such "learning" mechanisms are capable of creating programs.

In the usual situation, such learning systems are used to build "concepts", logical structures which represent or classify objects or sets of interest. As an illustration, suppose a system is given the following examples of good basketball players.

player 1: Height (Tall) and Hair (Brown)
player 2: Height (Tall) and Hair (Blond)

A learning system attempting to build a concept for good basketball players might use what Michalski (1980) calls the "dropping selector" rule for generalisation and hypothesise that the correct rule is Height (Tall). Such a generalisation might be too inclusive and later data might require restrictions as with (Height(Tall) and Reflex(Fast)). Thus the learning model is one of continuous adding of new information with the concept being enlarged or restricted further at each step to better match the incoming data. The eventual result of the learning process is a formula which correctly models all of the data samples, and if the samples are representative, all of the data in the domain.

Cohen & Sammut (1984) and Shapiro (1981) have shown how this kind of system can learn the concept of the example problem of this paper. We will follow the methodology of Cohen & Sammut in its solution. Assume the relationship to be learned is called "delete". The training process begins with an example of the target behaviour.

$x = \text{nil}, z = \text{nil}$

The learning system which has an array of transformation rules asserts that

$x = z$ and $x = \text{nil}$

and applies the dropping selector rule to hypothesise the following.

delete (x, z) is $x = z$

This is the system's first guess at the target input–output relation. It is clearly an over generalisation and is contradicted by the next piece of data.

$x = ((B)), z = \text{nil}$

This data requires a restriction on the current concept. The system retreats from the above hypothesis and proposes

$$\begin{aligned}\text{delete } (x, z) = (&x = \text{nil and } x = z) \text{ or}\\ &(\text{car}(x) = (B) \text{ and } \text{cdr}(x) = \text{nil}\\ &\quad \text{and } z = \text{nil})\end{aligned}$$

which can be generalised to

$$\begin{aligned}\text{delete } (x, z) = (&x = \text{nil and } x = z) \text{ or}\\ &(\text{not atom } (\text{car}(x)) \text{ and } \text{cdr}(x) = \text{nil}\\ &\quad \text{and } \text{cdr}(x) = z).\end{aligned}$$

The system continuously looks for opportunities to generalise by applying recursion. For example, here it could try the following form which, in fact, is a correct step toward the target.

$$\begin{aligned}\text{delete } (x, z) = (&x = \text{nil and } x = z) \text{ or}\\ &(\text{not atom } (\text{car}(x)) \text{ and}\\ &\quad \text{delete } (\text{cdr}(x), z))\end{aligned}$$

A series of additional examples could demonstrate the desired behaviour in the case of atom(car(x)) and result in the code

$$\begin{aligned}\text{delete } (x, z) = (&x = \text{nil and } x = z) \text{ or}\\ &(\text{not atom } (\text{car } x)) \text{ and}\\ &\quad \text{delete } (\text{cdr}(x), z)) \text{ or}\end{aligned}$$

Table 1. A summary of ten synthesis methodologies

Name	Reference	Synthesis method	Class of programs	Input information for sample program	Synthesised program
Strategic approach	Bibel & Hörnig (1984)	Heuristic search	General	Specification	$f(x) =$ if $x =$ nil then nil else $g(x)$ $g(x) =$ if atom(car(x)) then cons(car(x), f(cdr(x))) else f(cdr(x))
Divide and conquer	Smith (1985)	Input decomposition, computation and composition	Divide and conquer programs	Specification	$f(x) =$ if $x =$ nil then nil else append. (fxf). (car × cdr)x $f_1(x) =$ if atom(x) then list(x) else nil
Transformational methods	Broy (1983)	Search for successful sequence of transformations	General	Specification	$f(x) =$ if $x =$ nil then nil else append(f_1(car(x)), f(cdr(x))) $f_1(x) =$ if atom(x) then list(x) else nil
A deductive method	Manna & Waldinger (1980)	Search for successful sequence of transforms and resolutions	General	Specification	$f(x) =$ if $x =$ nil then nil else if not(atom(car(x))) then f(cdr(x)) else cons(car(x), f(cdr(x)))
Equational specification	Prywes *et al.* (1979)	Compilation	General	MODEL program	(Object code)

Function merging	Biermann (1978)	Decomposition into primitives and merging	Regular LISP programs	(*A* (*B*) *C*) yields (*A C*)	*f*(*x*) = cond(atom(*x*) *x*) (atom(car(*x*)) cons(*f*2(*x*), *f*3(*x*))) (*T f*(cdr(*x*))) *f*2(*x*) = *f*(car(*x*)) *f*3(*x*) = *f*(cdr(*x*))

From recurrence relations	Summers (1977)	Discovery of recurrence relations. Generalisation	Single loop LISP programs	Five examples	*f*(*x*) = if atom(*x*) then nil
					else
					if atom(car(*x*)) then
					cons(car(*x*), f(cdr(*x*)))
					else *f*(cdr(*x*))

A production rule methodology	Biermann & Smith (1979)	Diagnosis and application of production rules	LISP scanning programs	One example	*f*(*x*2) = cond(atom(*x*2) nil) (*T f*1(*x*2, *x*2)) *f*1(*x*1, *x*2) = cond(atom(car(*x*1)) *f*0(x1, x1, x2)) (*T f*(cdr(*x*2))) *f*0(*x*0, *x*1, *x*2) = cons(car(*x*0), *f*(cdr(*x*2)))
Generalisation on logical statements	Cohen & Sammut (1984)	Sequential generalisation and specialisation on concept model	General	Several examples	delete(*x*, *z*) = (*x* = nil and *x* = *z*) or (not atom(car(*x*)) and delete(cdr(*x*), *z*)) or (atom(car(*x*)) and car(*z*) = car(*x*) and delete(cdr(*x*), cdr(*z*)))
Natural language programming	Biermann (1983)	Parsing and interpretation of natural language	General	Two English sentences	Semantic representation of sentences

(atom (car(x))
and car(z) = car(x) and
delete (cdr(x), cdr(z)))

While this relationship is not a deterministic program of the style we seek, it is executable in the sense of PROLOG. Thus, it can be used as a synthesized program, or it could serve as a basis for a Manna–Waldinger derivation of a deterministic program.

Shapiro (1981) has given an elegant technique for doing learning of the kind described above and includes a proof that it converges on the correct target program if one exists.

5. Natural Language Programming

Rather than furnishing a formal specification or examples of the desired behaviour, a user might type or speak a description of the computation in natural language. For example, the user might say

"Read in an input list."
"Delete its nonatomic entries and
return the result."

Natural languages may be thought of as set manipulation languages in the programming environment. Processing generally begins at the head noun of the noun phrase where a set of objects within the area of focus for the dialogue are specified. Then modifiers to the head noun perform subsetting operations on the original set until a possibly complex subset is constructed. In the imperative sentence environment, such subsets are then passed to the execution routines for the imperatives for appropriate manipulation.

In the above two sentence sequence, the noun group in the first sentence is indefinite meaning in this environment that an object is being created conceptually. In English conversation, it is quite common to create conceptual objects with indefinite noun groups and then subsequently reference them with definite noun groups. The "read" imperative then seeks an object via a read operation to insert into the newly originated slot. The head noun "entries" in the second sentence finds all objects in focus which are entries. If other entries had been mentioned earlier in the conversation, it would probably still select the entries in the recently read list, because that list does ordinarily have entries and it is the primary object in focus. The modifier "nonatomic" selects out only a portion of the set of all entries and the resulting set is passed to the "delete" routine. The final clause addresses any singular object which can be designated a "result" which is in focus and outputs that object. Biermann (1983) has described a theory of natural language processing of the type explained here appropriate for programming applications.

Instead of stating the task in sequential imperative sentences, the user might have a conversation with the machine.

"Please write a delete routine for me."
WHAT WILL BE ITS NAME?
"Call it "delete"."
DESCRIBE ITS INPUTS.
and so forth.

Several such systems for interviewing the user about the target program and then doing the coding have been constructed by Balzer (1973), Green (1976), Heidorn (1972), and

Martin *et al.* (1974). These processors include natural language interfaces, knowledge bases for their respective domains, mechanisms for following conversation, code generation facilities, and numerous other features.

6. Conclusions

Table 1 gives a summary of the ten program synthesis methods discussed in this paper with their major characteristics. All of these approaches continue to be active research areas up to the present time.

Automatic programming is an important area of research both because it is a needed aid for people who want help in programming machines and because intelligent machines of the future should be programming themselves to better cope with their environments. Thus research in this field is fundamental to progress in artificial intelligence and will remain central to it for years to come. Some additional readings on this subject can be found in Barr & Feigenbaum (1982), Biermann (1976), Biermann & Guiho (1983), and Biermann *et al.* (1984).

References

Balzer, R. M. (1973). *A Global View of Automatic Programming.* Proc. of Third Joint Conference on Artificial Intelligence, pp. 494–499.

Barr, A., Feigenbaum, E. A. (eds) (1982). *Handbook of Artificial Intelligence,* Vol. 2. Los Altos: William Kaufmann, Inc.

Bibel, W., Hörnig, K. M. (1984). LOPS, a system based on a strategical approach to program synthesis. In: (Bierman, A., Guiho, G. & Kodratoff, Y., eds) *Automatic Program Construction Techniques,* pp. 69–89. New York: Macmillan.

Biermann, A. W. (1972). On the inference of Turing machines from sample computations. *Artif. Intell.* **3,** 181–198.

Biermann, A. W. (1976). Approaches to automatic programming. In: *Adv. Comput.,* **15,** pp. 1–63. New York: Academic Press.

Biermann, A. W. (1978). The inference of regular LISP programs from examples. *IEEE Trans. Systems, Man, Cybern.* Vol. SMC-8, pp. 585–600.

Biermann, A. W. (1983). Natural language programming. In: (Biermann, A. & Guiho, G., eds) *Computer Program Synthesis Methodologies,* pp. 335–368. Dordrecht: Reidl.

Biermann, A. W., Baum, R. I., Petry, F. E. (1975). Speeding up the synthesis of programs from traces. *IEEE Trans. Comput.* **C-24.**

Biermann, A. W., Guiho, G. (eds) (1983). *Computer Program Synthesis Methodologies.* Dordrecht: Reidel.

Biermann, A. W., Guiho, G., Kodratoff, Y. (eds) (1984). *Automatic Program Construction Techniques.* New York: Macmillan.

Biermann, A. W., Krishnaswamy, R. (1976). Constructing programs from example computations. *IEEE Trans. Software Eng.* **SE-2.**

Biermann, A. W., Smith, D. R. (1979). A production rule mechanism for generating LISP code. *IEEE Trans. Systems, Man, Cyber.* **SMC-9,** pp. 260–276.

Broy, M. (1983). Program construction by transformations: a family tree of sorting programs. In: (Biermann, A. & Guiho, G., eds) *Computer Program Synthesis Methodologies,* pp. 1–49. Dordrecht: Reidel.

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

Cohen, B., Sammut, C. (1984). Program synthesis through concept learning. In: (Biermann, A., Guiho, G. & Kodratoff, Y., eds) *Automatic Program Construction Techniques,* pp. 463–482. New York: Macmillan.

Gerhart, S. L. (1976). Proof theory of partial correctness verification systems. *SIAM J. Comp.* **5,** 355–377.

Green, C. (1976). The design of the PSI program synthesis system. *Proceedings of the Second International Conference on Software Engineering.*

Heidorn, G. (1972). *Natural Language Inputs to a Simulation Programming System.* Monterey, California: Technical Report, Naval Postgraduate School.

Kodratoff, Y., Jouannaud, J.-P. (1984). Synthesizing LISP programs working on the list level of embedding. In: (Biermann, A., Guiho, G. & Kodratoff, Y., eds) *Automatic Program Construction Techniques,* pp. 325–374. New York: Macmillan.

Manna, Z., Waldinger, R. (1979). Synthesis: dreams ⇒ program. *IEEE Trans. Software Eng.*, **SE-5**, 294–328.
Manna, Z., Waldinger, R. (1980). A deductive approach to program synthesis. *ACM Trans. Programming Languages and Systems* **2**, 90–121.
Martin, W. A., Ginzberg, M. J., Krumland, R., Mark, B., Morgenstern, M., Niamir, B., Sunguroff, A. (1974). Internal Memos, Automatic Programming Group. Cambridge, Mass.: Massachusetts Institute of Technology.
Michalski, R. S. (1980). Pattern recognition as rule-guided inductive inference. *IEEE Trans. Pattern Analysis Machine Intell.*
Pnueli, A., Prywes, N. S., Zarhi, R. (1984). Scheduling equational specifications and non-procedural programs. In: (Biermann, A., Guiho, G. & Kodratoff, Y., eds) *Automatic Program Construction Techniques*, pp. 273–288. New York: Macmillan.
Prywes, N. S., Pnueli, A., Shastry, S. (1979). Use of a non-procedural specification language and associated program generator in software development. *ACM Trans. Programming Languages and Systems* **1**, 196–217.
Shapiro, E. Y. (1981). Inductive inference of theories from facts. Report 192. New Haven: Department of Computer Science, Yale University.
Smith, D. R. (1985). Top-down synthesis of simple divide and conquer algorithms. *Artif. Intell.* (in press).
Summers, P. D. (1977). A methodology for LISP program construction from examples. *J. Assoc. Comp. Mach.* **24**, 161–175.