# SYNTHESIS: DREAMS ⟹ PROGRAMS

Technical Note 156

November 1977

By: Zohar Manna
Artificial Intelligence Lab
Stanford University

Richard Waldinger
Artificial Intelligence Center
SRI International

Research sponsored by:

National Science Foundation
Office of Naval Research
Advanced Research Projects Agency

Approved by:

Peter E. Hart, Director
Artificial Intelligence Center

Earle D. Jones, Executive Director
Information Science and Engineering Division

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

This research was supported in part by the National Science Foundation under Grants DCR72-03737 A01 and MCS76-83655, by the Office of Naval Research under Contracts N00014-76-C-0687 and N00014-75-C-0816, by the Advanced Research Projects Agency of the Department of Defense under Contract MDA903-76-C-0206, and by a grant from the United States-Israel Binational Science Foundation (BSF).

**Abstract:**

Deductive techniques are presented for deriving programs systematically from given specifications. The specifications express the purpose of the desired program without giving any hint of the algorithm to be employed. The basic approach is to transform the specifications repeatedly according to certain rules, until a satisfactory program is produced. The rules are guided by a number of strategic controls. These techniques have been incorporated in a running program-synthesis system, called DEDALUS.

Many of the transformation rules represent knowledge about the program's subject domain (e.g., numbers, lists, sets); some represent the meaning of the constructs of the specification language and the target programming language; and a few rules represent basic programming principles. Two of these principles, the *conditional–formation rule* and the *recursion–formation rule*, account for the introduction of conditional expressions and of recursive calls into the synthesized program. The termination of the program is ensured as new recursive calls are formed.

Two extensions of the recursion-formation rule are discussed: a *procedure–formation rule*, which admits the introduction of auxilliary subroutines in the course of the synthesis process, and a *generalization rule*, which causes the specifications to be altered to represent a more general problem that is nevertheless easier to solve. Special techniques are introduced for the formation of programs with side effects.

The techniques of this paper are illustrated with a sequence of examples of increasing complexity; programs are constructed for list processing, numerical calculation, and array computation.

The methods of program synthesis can be applied to various aspects of programming methodology -- program transformation, data abstraction, program modification, and structured programming.

The DEDALUS system accepts specifications expressed in a high-level language, including set notation, logical quantification, and a rich vocabulary drawn from a variety of subject domains. The system attempts to transform the specifications into a recursive, LISP-like target program. Over one hundred rules have been implemented, each expressed as a small program in the QLISP language.

## Contents:

# INTRODUCTION

In recent years there has been increasing activity in the field of program verification. The goal of these efforts is to construct computer systems for determining whether a given program is correct, in the sense of satisfying given specifications. These attempts have met with increasing success; while automatic proofs of the correctness of large programs may be a long way off, it seems evident that the techniques being developed will be useful in practice, to find the bugs in faulty programs and to give us confidence in correct ones.

The general scenario of the verification system is that a programmer will present his completed computer program, along with its specifications and associated documentation, to a system which will then prove or disprove its correctness. It has been pointed out, most notably by the advocates of structured programming, that this is "putting the cart before the horse." Once we have techniques for proving program correctness, why should we wait to apply them until after the program is complete? Instead, why not ensure the correctness of the program while it is being constructed, thereby developing the program and its correctness proof "hand in hand"?

The point is particularly well-taken when we consider that program verification relies on automatic theorem-proving techniques. These techniques embody principles of deductive reasoning, the same principles that are applied by a programmer in constructing the program in the first place. Why not employ these principles in an *automatic synthesis system*, which would construct the program instead of merely proving its correctness? Granted, to construct a program requires more originality and creativeness than to prove its correctness, but both tasks require the same kind of thinking.

Structured programming itself made an early contribution to the automatic synthesis of computer programs in laying down principles for the development of programs from their specifications. These principles are intended to serve as guidelines to be followed by a human programmer. However, they are not formulated precisely enough to be carried out by a machine. Indeed, the proponents of structured programming have been most pessimistic about the possibility of ever automating their techniques; Dijkstra has gone so far as to say that we shouldn't automate programming even if we can, because we would take away all our enjoyment of the task.

Programming is a challenging task, and its automation is a part of artificial intelligence. A system to construct computer programs must have a broad range of knowledge about programming languages, programming techniques, and the subject domain of the program to be constructed. Furthermore, it must have the ability to retrieve the relevant components of its knowledge and to combine them to perform the task at hand. Programming is among the most demanding human activities, and is among the last tasks computers will do well. Nevertheless, the intrinsic interest and practical importance of the programming task have motivated many researchers to consider the possibility of automating it.

Several years ago, we began our research on automatic program synthesis by considering a large number of simple programming tasks. In examining the derivations of programs to achieve these tasks, we observed certain regularities, steps that are performed over and over again in a variety of subject domains, and that therefore can be regarded as representing basic programming principles. We have specified these principles precisely, and have applied them to the construction of less trivial programs.

In this paper, we present some of the basic principles to be incorporated into an automatic program-synthesis system. Such a system accepts specifications that express the purpose of the program to be constructed, without giving any hint of the algorithm to be employed. With no further human intervention, the system attempts to transform these specifications into a program that achieves the expressed purpose. This program is guaranteed to be correct and will always terminate; for the most part, we will not be concerned with its efficiency.

The specifications are expressed in a *specification language* rich with constructs from the subject domain of the application. Because the specification language does not need to be executed, it can afford high-level constructs close to our way of thinking about the subject. Specifications represented in such a language are likely to be easy to formulate and to correspond correctly to our intentions. The details of the particular *target language*--the language in which the program is to be constructed--are not important. In our examples, we employ a simple LISP-like language.

Our basic approach is to transform the specifications repeatedly according to certain *transformation rules*. Guided by a number of strategic controls, these rules attempt to produce an equivalent description composed entirely of constructs from the target language. Many of the transformation rules represent knowledge about the program's subject domain; some explicate the constructs of the specification and target languages; and a few rules represent basic programming principles.

Some of these techniques have been incorporated into an experimental program-synthesis system called DEDALUS (the DEDuctive ALgorithm Ur-Synthesizer). The purpose of this system is not to be applied in practice but rather to test our program-synthesis ideas. Most of the examples included in this paper have been carried out by the DEDALUS system. However, the emphasis of the paper is not on the details of the DEDALUS implementation, but on the basic programming principles it incorporates, which can be applied in any system.

In the past few years, there have appeared several varieties of programming methodology, e.g., structured programming, program transformation, and data abstraction. These disciplines recommend systematic approaches to program construction for making the programming process simpler and more reliable. The techniques of program synthesis serve to facilitate the application of each of these disciplines. In this way, program-synthesis research can be of value long before its ultimate goal is achieved.

In this paper, we present the basic concepts and principles of program synthesis, we extend these methods to allow the synthesis of programs with side effects, and we apply these techniques to various aspects of programming methodology. Historical remarks, comparisons with other approaches to automatic programming, and notes on the DEDALUS implementation are reserved for a final section.

# 1. CONCEPTS

## A. Specifications

The first requirement of a specification language is that it should allow us to express the purpose of the desired program directly. In other words, once we have formed a precise idea of what the program is intended to do, we should be able to formulate the specifications immediately, without paraphrase. Furthermore, it should be easy for the programmer and other people to read and understand the specifications and to see that they are correct.

For this reason, it is necessary that the specification language contain very high-level constructs, which correspond to the concepts we use in thinking about the problem and which are endemic to the subject domain of the target program. Such constructs are typically not included in a conventional programming language, because it may be impossible to find a uniform way of computing them or because they may not be amenable to efficient implementation.

Because a specification language should have a large number of constructs, and because these constructs are particular to the subject domain, we do not attempt to define a complete specification language. Instead, we present the specifications of some of the programs we will use as examples later in this paper, to illustrate some of the most useful constructs.

Suppose we want to construct a program, called *lessall*, to test whether a given number $x$ is less than every member of a given list $l$ of numbers, and to output *true* or *false* accordingly. This program can be described as

```
lessall(x l) <== compute x < all(l)
            where   x is a number and
                    l is a list of numbers .
```

Here, the expression $x < all(l)$ denotes the condition that $x$ is less than every member of the list $l$; its value is *true* or *false* depending on whether or not the condition holds. The expression **compute . . .** is the *output specification*; it provides a description of the output the target program is intended to produce. The expression **where . . .** is the *input specification*; it gives the conditions the inputs $x$ and $l$ can be expected to satisfy.

To specify a program *maxlist* to compute the largest element of a given list $l$, we write

```
maxlist(l) <== compute some z : z ∈ l and z ≥ all(l)
            where l is a nonempty list of numbers.
```

Here, the construct "*some* $z \in P(z)$" denotes any element $z$ satisfying the condition $P(z)$, and $u \in v$ means that $u$ is a member of the list (or set) $v$.

Another example: the greatest common divisor (*gcd*) of two nonnegative integers is the largest integer that divides both of them. To specify a program to compute the *gcd* of $x$ and $y$, we write

$gcd(x\ y)$ <== **compute** $max\{z : z|x \text{ and } z|y\}$
**where** $x$ and $y$ are nonnegative integers and
$x \neq 0$ or $y \neq 0$ .

Here, $max\ S$ is the largest element of the set $S$. The input condition $x \neq 0$ or $y \neq 0$ is included because if both $x$ and $y$ are zero, then any integer divides each of them, and the set of all their common divisors is infinite and has no largest element.

The Cartesian product *cart* of two sets $s$ and $t$ is the set of all pairs whose first element belongs to $s$ and whose second element belongs to $t$; a program to compute it is specified by

$cart(s\ t)$ <== **compute** $\{\ (x\ y) : x \in s \text{ and } y \in t\ \}$
**where** $s$ and $t$ are finite sets.

Here, $(x\ y)$ denotes the pair whose elements are $x$ and $y$ .

## B. The Target Language

The techniques we employ in this paper are not dependent on the particular choice of a *target language*, the language in which the desired program is to be expressed. However, for the sake of definiteness, we will represent the target programs in this paper in a fixed, LISP-like language, which should be readily understandable.

For numbers, the target language includes such familiar operations as $x + y$, $x - y$, $x \leq y$, etc. For lists, we assume that the target language contains the usual LISP primitives:

*head*$(l)$ : the first element of the nonempty list $l$

*tail*$(l)$ : the list of all but the first element of the nonempty list $l$

*cons*$(x\ l)$ : the list formed by inserting the element $x$ at the beginning of the list $l$ .

Furthermore, we include the common *conditional expression*:

*if* $P$ *then* $x$ *else* $y$ : $x$ if $P$ is true,
$y$ if $P$ is false.

Finally, we employ *recursion*; for example, a program $f(l)$ may be defined in terms of a recursive call $f(tail(l))$.

Of course, we can use any of the target-language constructs in formulating the specifications. Thus, the target-language may be considered to be a subset of the specification language.

A segment of a program description that consists entirely of target-language constructs will be called a *primitive segment.*

At times we will choose to add new primitives to the target language. Thus, if we want to write a program in a new subject domain, we will add the primitives appropriate to that domain. If we want to express a program in terms of some given set of procedures, we will treat those procedures as primitives. In the section on side effects (Section 4), we will include constructs such as assignment statements and arrays in the target language.

By the same token, for certain tasks we may choose to delete primitives from the target language. For instance, to construct a more efficient program we may delete certain time-consuming primitives. The DEDALUS system allows the user to add or delete constructs from its primitive set for a particular task.

## C. Transformation Rules

Our basic approach to program synthesis is to employ a large number of *transformation rules*, which replace one segment of a program description by another, equivalent description. The task of program synthesis is then reduced to applying these rules to the given specification repeatedly until a primitive program is produced.

Some transformation rules express the principles of the underlying semantic domain (e.g., the properties of the integers or list structures). Other rules express the meaning of the constructs in the specification and the target languages (e.g., $\{u : P(u)\}$ in the specification language and $head(l)$ in the target language). Still others represent a formulation of basic programming techniques, which do not depend on a particular subject domain (e.g., the introduction of conditional expressions and recursion).

We use the notation

$$t \Rightarrow t'$$

to denote a transformation rule that an expression of form $t$ may be replaced by the corresponding expression $t'$. The transformation may be applied to any subexpression $t$ of the current program description. It is not to be applied in the reverse direction unless another rule of form $t' => t$ is given explicitly.

For example, the rule

$$\mathit{true\ and\ Q} => Q$$

means that any expression of form *true and Q* may be replaced by $Q$. By applying this rule, we may replace a program description

$$\mathit{max}\{z : \mathit{true\ and}\ z|y\}$$

by the description

$$\mathit{max}\{z : z|y\}\ .$$

A rule

$$t => t' \quad \text{if } P$$

denotes that the transformation $t => t'$ can be applied only if the condition $P$ is true. Thus the rule

$$u|v => \mathit{true} \quad \text{if } u \text{ is an integer and } v = 0$$

denotes that a program segment $u|v$ can be replaced by *true* if $u$ is known to be an integer and $v$ to be zero whenever the segment is executed. Thus, this rule can be applied to transform a program description

```
if y = 0
then x|y
else . . .
```

into

```
if y = 0
then true
else . . . ,
```

where $x$ is known to be an integer.

Often, more than one rule can be applied to the same program description or even to the same segment. For example, the logical rule

$$P \textit{ and } Q \Rightarrow Q \textit{ and } P$$[1]

and the numerical rule

$$u|v \textit{ and } u|w \Rightarrow u|v \textit{ and } u|w-v \quad \text{if } u, v, \text{ and } w \text{ are integers}$$

can both be applied to the program description

$$max\{z : z|x \textit{ and } z|y\}.$$

In such cases it must be decided which rule is best to apply. This difficult problem must be faced in any transformation-rule system. We prefer to postpone such considerations until after we have presented some concrete examples. (See Section 2D on "Strategic Controls.")

## D. Derivation Trees

In applying a transformation rule to a given program description, we obtain a new program description, which we regard as a subgoal of the first. To this subgoal we apply additional transformation rules repeatedly, until a primitive program description is obtained. This description is the desired program.

The top-level goal is obtained directly from the program's specifications. Thus, if the program $f$ is specified by

$f(x)$ <== **compute** $P(x)$
**where** $Q(x)$,

the top-level goal will be

**Goal:** **compute** $P(x)$.

(Here, $Q(x)$ is a condition but $P(x)$ may be any expression in the specification language.) For example, in deriving the *gcd* program, we are given the specifications

*gcd*$(x\ y)$ <== **compute** $max\{z : z|x \textit{ and } z|y\}$
**where** $x$ and $y$ are nonnegative integers and
$x \neq 0$ or $y \neq 0$.

---------------------

[1] Actually, the DEDALUS system does not use this rule explicitly; the same effect is achieved by a different mechanism. See "Implementation," Section 6B.

Our top-level goal is thus

**Goal 1:** **compute** $max\{z : z|x \text{ and } z|y\}$ .

By applying the transformation rule

$P \text{ and } Q \Rightarrow Q \text{ and } P$ ,

we obtain

**Goal 2:** **compute** $max\{z : z|y \text{ and } z|x\}$ .

If a transformation rule imposes a condition $P$, which must be true if the rule is to be applied, a subgoal of the form

**Goal:** **prove** $P$

must be achieved before the rule can be applied. For example, in developing the program *lessall*($x$ $l$) to test if a number is less than every element of a list of numbers, we begin with the top-level goal

**Goal 1:** **compute** $x < all(l)$ .

In attempting to apply the rule

$P(all(l)) \Rightarrow true$ if $l$ is the empty list,

which states that any property $P$ holds for every element of the empty list, we generate the subgoal

**Goal 2:** **prove** $l$ is the empty list .

To accomplish such a task, we must apply transformation rules repeatedly to the expression to be proved, until the expression *true* is produced. If, instead, *false* is produced, or if we encounter a situation in which no rule can be applied, the goal of proving $P$ is aborted, and the attempt to use the rule that imposed $P$ as a condition is abandoned.

If no rule applies to a given subgoal, *backtracking* occurs; we seek alternate rules to apply to previous subgoals. Backtracking will be discussed further in the section on "Strategic Controls" (Section 2D).

By the process we have just outlined, a tree of goals and subgoals is generated. We will call this structure a *program derivation tree.*

# 2. ELEMENTARY PROGRAMMING PRINCIPLES

## A. The Formation of Conditional Expressions

To illustrate the formation of conditional expressions and recursive calls, we exploit a single simple example. The program to be constructed, $lessall(x\ l)$, is intended to test whether a given number $x$ is less than every member of a given list $l$ of numbers, and to output *true* or *false* accordingly. The specifications, as indicated in Section 1A, can be expressed as

$lessall(x\ l)$ <== **compute** $x < all(l)$
    **where** $x$ is a number and
        $l$ is a list of numbers .

Note that the output description uses the *all* specification construct, which is not primitive; therefore, we attempt to apply transformation rules to paraphrase the output description using only primitive constructs of the target language.

We assume we have at our disposal two rules that explicate the *all* construct:

- The *vacuous rule*

  $P(all(l)) \Rightarrow true$  if $l$ is the empty list

says that any property is true of every element of the empty list.

- The *decomposition rule*

  $P(all(l)) \Rightarrow P(head(l))\ and\ P(all(tail(l)))$  if $l$ is a nonempty list

states that a property holds for every element of a nonempty list if it holds for the first element and for all the rest.

Our top-level goal is formed directly from the program's specifications:

**Goal 1:** **compute** $x < all(l)$ .

In this discussion we will not consider how to select the rule to be applied; we will assume for the time being that the appropriate rule magically appears when it is relevant.

One transformation rule that applies to the current output description is the vacuous rule,

$P(all(l)) \Rightarrow true$  if $l$ is the empty list .

This rule would allow us to reduce our output description to *true* if only we could achieve the subgoal

**Goal 2:** **prove** $l$ is the empty list .

Of course, we cannot prove or disprove this condition: $l$ is an input that is known to be a list, but that may or may not be empty. This is an occasion for applying the *conditional–formation rule.*

> Conditional expressions are introduced into programs as a result of hypothetical reasoning during the program-formation process. If we fail to prove or disprove a subgoal of the form
>
> **prove** $P$ ,
>
> the *conditional–formation rule* allows us to introduce a case analysis and consider separately the case in which $P$ is true and $P$ is false. Suppose we succeed in constructing a program segment $s_1$ that solves our problem under the assumption that $P$ is true, and another segment $s_2$ that solves the problem under the assumption that $P$ is false. Then we combine the two segments into a conditional expression
>
> *if* $P$ *then* $s_1$ *else* $s_2$ ,
>
> which solves the problem regardless of whether $P$ is true or false. Note that to ensure that this expression is primitive, we apply the conditional–formation rule only when $P$ itself is a primitive logical statement.

Let us return to our example. Having failed to prove Goal 2, that $l$ is empty, we attempt to construct a program segment that will solve our problem under the assumption that $l$ is empty.

**Case** $l$ is empty: In this case, we are justified in applying the vacuous rule

$P(all(l))$ => *true* if $l$ is the empty list,

to Goal 1, **compute** $x < all(l)$, yielding the primitive program segment *true*. This segment solves our problem in this case.

We have yet to consider the case in which $l$ is nonempty. This requires the formation of a recursive call, which will be discussed in the next section. However, at this point, we know that the program will have the form

```
lessall(x l) <== if empty(l)
                 then true
                 else . . . .
```

> Case analysis in theorem proving has been emphasized by Bledsoe and Tyson [1977]. Other program-synthesis systems that form conditional expressions by case analysis have been implemented by Luckham and Buchanan [1974] and Warren [1976].

## B. The Formation of Recursive Calls

We illustrate the formation of recursive calls by continuing the construction of the *lessall* program. Recall that it remains to consider the case in which $l$ is a nonempty list.

**Case** $l$ is nonempty: In this case we fail to achieve Goal 2, to prove that $l$ is empty, and therefore we look for some alternate means for approaching Goal 1, **compute** $x < \mathit{all}(l)$.

Another rule that applies to Goal 1 is the *all* decomposition rule

$$P(\mathit{all}(l)) \Rightarrow P(\mathit{head}(l)) \text{ and } P(\mathit{all}(\mathit{tail}(l))) \quad \text{if } l \text{ is a nonempty list .}$$

This rule imposes the condition

**Goal 3:** **prove** $l$ is a nonempty list,

which is satisfied immediately because we have assumed in our case analysis that $l$ is nonempty. The rule, therefore, transforms Goal 1 into

**Goal 4:** **compute** $x < \mathit{head}(l)$ *and* $x < \mathit{all}(\mathit{tail}(l))$.

To compute the truth value of $x < \mathit{head}(l)$ is simple, because $x$ and $l$ are inputs, and *head* is a primitive construct. It remains, therefore, to achieve

**Goal 5:** **compute** $x < \mathit{all}(\mathit{tail}(l))$ .

Note that this subgoal is an instance of our original Goal 1, to compute $x < \mathit{all}(l)$, with inputs $x$ and $l$ replaced by $x$ and $\mathit{tail}(l)$. This is an opportunity for applying the *recursion–formation rule.*

In general, suppose we are to develop a program whose specifications are of form

$f(x)$ <== **compute** $P(x)$<br>
**where** $Q(x)$ ,

in which $Q(x)$ is a condition but $P(x)$ may be any expression in the specification language. Assume we encounter a subgoal

**compute** $P(t)$

that is an instance of the output specification **compute** $P(x)$. Then we can attempt to achieve this subgoal by forming a recursive call $f(t)$, because the program $f(x)$ is intended to compute $P(x)$ for any $x$ that satisfies $Q(x)$. To ensure that the introduction of this recursive call is legitimate, we must verify two conditions:

- The *input condition*, $Q(t)$, which establishes that the argument $t$ of the recursive call $f(t)$ satisfies the required input condition of the desired program; otherwise, the program $f$ is not guaranteed to yield the expected output.

- A *termination condition*, which ensures that the recursive call cannot cause an infinite computation. A recursive call can fail to terminate if its execution leads to another recursive call, which leads to another, and so on indefinitely.

The termination condition is expressed in terms of the "well-founded set" concept, which will be explained in a later section devoted exclusively to termination. In the meantime, we will appeal to intuitive arguments to establish termination.

Note that to ensure that the recursive call $f(t)$ be primitive, we apply the recursion-formation rule only when the argument $t$ itself is primitive.

Let us return to our example. The recursion-formation rule observes that Goal 5, to compute $x < all(tail(l))$, is an instance of our output specification, $x < all(l)$, with inputs $x$ and $l$ replaced by $x$ and $tail(l)$; therefore it proposes that we achieve this goal with a recursive call $lessall(x\ tail(l))$. For this purpose, the rule imposes two conditions, the input condition

**Goal 6:** **prove** $tail(l)$ is a list,

and the termination condition

Goal 7: prove *lessall*(*x tail*(*l*)) terminates.

The input condition that *tail*(*l*) is a list can be proved directly by invoking a transformation

*tail*(*l*) is a list => *true* if *l* is a list ,

a basic rule describing list structures. To achieve the termination condition is also straightforward, because the argument *tail*(*l*) of the recursive call is a proper sublist of the input *l*; therefore only a finite number of recursive calls can occur before the second argument is reduced to the empty list. Consequently, we are permitted to introduce a recursive call *lessall*(*x tail*(*l*)) at this point. This satisfies Goal 5; Goal 4 is then satisfied by the program segment *x* < *head*(*l*) *and lessall*(*x tail*(*l*)). This segment is composed entirely of primitive constructs of our target language.

We have succeeded in finding primitive program segments that solve our problem in both cases, whether *l* is empty or not. Therefore the conditional-formation rule combines the two program segments into a conditional expression. The final program is

```
lessall(x l) <== if empty(l)
                 then true
                 else x < head(l) and lessall(x tail(l)) .
```

The above technique causes the formation of a recursive program. If we are working in a target language that does not admit recursion, it is necessary to transform the program further, to replace the recursion by another repetitive construct. In many cases, a recursive program can be transformed into an iterative program of comparable complexity. In the worst case, we can always replace a recursive procedure with an iterative equivalent by the explicit introduction of a stack.

The above recursion-formation rule is the same as the "folding" rule of the Burstall and Darlington [1977] system for the transformation of recursive programs. Their system does not check the input and termination conditions.

## C. Termination

In the preceding example we relied on intuitive arguments to establish the termination of the program we constructed. In fact, for that example, the termination argument was quite straightforward. In this section, we will consider a general mechanism for proving the termination of a recursive program at the same time as it is being constructed. We will illustrate this mechanism with an example for which the termination proof is somewhat more subtle.

The program we construct is intended to compute the greatest common divisor, $gcd(x\ y)$, of two nonnegative integers $x$ and $y$. The specifications, as indicated in Section 1A, are expressed as

$$gcd(x\ y) \Leftarrow\!\!= \textbf{compute}\ max\{z : z|x\ \mathit{and}\ z|y\}$$
$$\textbf{where}\ x\ \text{and}\ y\ \text{are nonnegative integers and}$$
$$x \neq 0\ \text{or}\ y \neq 0\,.$$

Recall that the input condition $x \neq 0$ or $y \neq 0$ is imposed because the $gcd$ is not defined when both its arguments are zero.

The output specification is expressed in terms of the set constructor $\{u : P(u)\}$, which is not primitive. We therefore attempt to transform it into an equivalent primitive description.

We assume that the following rules about the integers are included among the transformations of our system:

$$u|v \Rightarrow \mathit{true} \quad \text{if } v = 0$$

(every integer divides 0),

$$u|v\ \mathit{and}\ u|w \Rightarrow u|v\ \mathit{and}\ u|w-v$$

(the common divisors of $v$ and $w$ are the same as those of $v$ and $w-v$), and

$$max\{u : u|v\} \Rightarrow v \qquad \text{if } v \text{ is a positive integer}$$

(every positive integer is its own greatest divisor).

As usual, our first goal is derived directly from the output specification:

**Goal 1:** **compute** $max\{z : z|x\ \mathit{and}\ z|y\}$ .

There are at least two rules that match the subexpression $z|x\ \mathit{and}\ z|y$; they are the logical rule

$$P\ \mathit{and}\ Q \Rightarrow Q\ \mathit{and}\ P$$

and the numerical rule

$$u|v\ \mathit{and}\ u|w \Rightarrow u|v\ \mathit{and}\ u|w-v\ .$$

Either rule will lead to a successful program; suppose we attempt the logical rule first. Then we develop the subgoal

**Goal 2:** **compute** $max\{z : z|y \text{ } and \text{ } z|x\}$ .

Goal 2 is an instance of Goal 1 itself, with $x$ and $y$ replaced by $y$ and $x$; therefore, the recursion-formation rule attempts to satisfy Goal 2 with a recursive call *gcd*($y$ $x$). To ensure that this step is legitimate, the rule imposes an input condition

**Goal 3:** **prove** $y$ and $x$ are nonnegative integers and
$y \neq 0$ or $x \neq 0$

obtained by replacing $x$ and $y$ by $y$ and $x$, respectively, in the input condition of the specification. This condition is easily established, because it is an equivalent form of the given input condition itself. Furthermore, the recursion-formation rule imposes a termination condition, to ensure that the proposed recursive call terminates :

**Goal 4:** **prove** *gcd*($y$ $x$) terminates.

We will begin by attempting to use the same sort of informal argument we employed in the previous example proving the termination of this recursive call. Later in this example, we will be forced to introduce the more formal and general apparatus. To establish termination, it suffices to achieve

**Goal 5:** **prove** $y < x$ ,

because $x$ and $y$ are both known to be nonnegative integers (by the input condition), and because $y$ is the first argument of the recursive call.

If we establish Goal 5, only a finite sequence of recursive calls can occur before the first argument is reduced to zero. However, we cannot prove or disprove Goal 5; $x$ and $y$ are both input variables, and we have no way of knowing if one of them is bigger than the other. As before, the conditional-formation rule causes a case analysis to be introduced.

**Case** $y < x$ : Here, both the input condition and the termination condition for introducing the recursive call *gcd*($y$ $x$) are satisfied. We have thus completed one branch of the case analysis; we have yet to consider the alternate case. However, at this stage we know that the final program will have the form

```
gcd(x y) <== if y < x
             then gcd(y x)
             else . . . .
```

**Case** $x \le y$ : Here, it is not legitimate to introduce the recursive call $gcd(y\ x)$ to achieve Goal 2, because the termination condition is not satisfied. Assuming that no other rules succeed in reducing Goal 2 to a primitive segment, we are led to consider alternate means of achieving the original Goal 1 in this case.

Recall that among other rules that applied to Goal 1 was the numerical rule

$u|v \text{ and } u|w \Rightarrow u|v \text{ and } u|w-v$ .

This rule causes the generation of a new goal

**Goal 6:** **compute** $max\{z : z|x \text{ and } z|y-x\}$ .

This goal has the same form as the original Goal 1, but with the inputs $x$ and $y$ replaced by $x$ and $y-x$; the recursion-formation rule suggests satisfying Goal 6 with the recursive call $gcd(x\ y-x)$ .

To ensure that the arguments $x$ and $y-x$ are legitimate, the rule imposes the input condition

**Goal 7:** **prove** $x$ and $y-x$ are nonnegative integers and
$x \ne 0$ or $y-x \ne 0$ ;

to guarantee that the proposed recursive call will terminate, the rule also imposes the termination condition

**Goal 8:** **prove** $gcd(x\ y-x)$ terminates.

Let us examine Goal 7 first: that $x$ and $y-x$ are nonnegative integers follows from the original input specification and the case assumption $x \le y$; the condition

$x \ne 0$ or $y-x \ne 0$

leads us to attempt to prove either

**Goal 9:** **prove** $x \ne 0$ ,

or

**Goal 10:** **prove** $y-x \ne 0$ .

We fail to prove or disprove Goal 9; therefore, the conditional-formation rule introduces a case analysis.

**Case** $x \neq 0$ : Here, the input condition for the proposed recursive call $gcd(x\ y-x)$ is satisfied; it remains to show the termination condition (Goal 8).

If this were the only recursive call in the entire program, its termination would be easy to establish. After all, we know in this case that $x$ is a positive integer and that $y-x$ is a nonnegative integer; furthermore, $y-x$ is strictly less than the second input $y$. Thus, each execution of this recursive call reduces the second argument, and only a finite number of executions can occur before the second argument is reduced to zero. However, the program we are developing already contains another recursive call $gcd(y\ x)$; we must consider the possibility that an infinite computation involving both recursive calls might occur.

This is a real possibility, because the recursive call $gcd(y\ x)$ actually increases the second argument. We therefore must treat both recursive calls at once, and this requires a more sophisticated mechanism for proving termination conditions.

> In general, to prove termination we employ the concept of a *well-founded set*, one whose elements are ordered in such a way that no infinite decreasing sequence of elements can exist. For example, the nonnegative integers, under the usual less-than ordering, constitute a well-founded set, whereas the entire set of integers does not.
>
> To prove the termination of a recursive program $f(x)$ with recursive calls $f(t_1)$, $f(t_2)$, $\ldots$, $f(t_n)$, we show that $x$, $t_1$, $t_2$, $\ldots$, $t_n$ all belong to some well-founded set $W$, ordered by a relation $\prec$, and that
>
> $$t_1 \prec x\ ,\ t_2 \prec x\ ,\ .\ .\ .\ ,\ \text{and } t_n \prec x\ .$$
>
> This condition suffices to ensure termination, because if there were a nonterminating computation, it would contain an infinite sequence of recursive calls, whose arguments would constitute an infinite decreasing sequence in the well-founded set. But a well-founded set contains no infinite decreasing sequences.
>
> By the method we have just described, to establish the termination of a program $f(x)$ with many recursive calls $f(t_1)$, $f(t_2)$, $\ldots$, $f(t_n)$, we must show that each argument $t_i$ is less than the original input $x$ under a single well-founded ordering $\prec$. This implies that, during the synthesis of the program, whenever we introduce a new recursive call $f(t_i)$ we must show that $t_i \prec x$ under the same ordering $\prec$ which we have used to establish the termination of the recursive calls $f(t_1)$, $f(t_2)$, $\ldots$, $f(t_{i-1})$, introduced previously. If we

cannot, we must modify the well-founded set $W$ and the ordering $\prec$ so that $t_i \prec x$, while ensuring that the relations $t_1 \prec x$, $t_2 \prec x$, $\ldots$, $t_{i-1} \prec x$ are still satisfied.

If the program has more than one argument, the ordering $\prec$ of the well-founded set may need to compare pairs or tuples of arguments. For this purpose it is convenient to use the lexicographic ordering between tuples. For pairs of nonnegative integers, for example, this ordering is defined as follows:

$$(x_1\ x_2) \prec (y_1\ y_2) \quad \text{if } x_1 < y_1 \text{, or if } x_1 = y_1 \text{ and } x_2 < y_2 \,.$$

Thus, the second components are ignored unless the first components are equal. This lexicographic ordering can be shown to be well-founded: there exist no infinite sequences of pairs of nonnegative integers that decrease under this ordering. A general notion of lexicographic ordering on arbitrary tuples of elements can be defined in a similar way.

In the *gcd* example, we have already proved the termination condition of the recursive call $gcd(y\ x)$ by showing that the first argument $y$ of the recursive call is less than the first input $x$; in other words, we have used the ordering $\prec$ defined by

$$(u_1\ u_2) \prec (v_1\ v_2) \quad \text{if } u_1 < v_1 \,.$$

This is a well-founded ordering between pairs of nonnegative integers. Thus, in proving the termination condition for the proposed new recursive call $gcd(x\ y-x)$, we attempt to show that

$$(x\ y-x) \prec (x\ y)$$

under this ordering, i.e., that $x < x$. This attempt fails; the first argument is not reduced by the proposed recursive call. We therefore try to modify the ordering $\prec$ to establish the termination condition for the second recursive call as well.

The first argument $x$ of the proposed recursive call $gcd(x\ y-x)$ is nonnegative and is identical to the first input $x$; we have also seen that the second argument $y-x$ is a nonnegative integer (since we have assumed that $x \le y$) and is less than the second input $y$ (since $x$ is positive in this case).

This suggests that we modify the ordering $\prec$ to be the lexicographic ordering. This ordering will allow us to prove the termination conditions for both recursive calls.

The use of the recursive call $gcd(x\ y-x)$ has been justified in this case, because its input condition (Goal 7) and its termination condition (Goal 8) have been established. The partial program we have constructed so far is

```
gcd(x y) <== if y < x
             then gcd( y x)
             else if x ≠ 0
                  then gcd(x y−x)
                  else . . . .
```

We have yet to consider the case in which $x = 0$ .

**Case $x = 0$ :** In this case, the recursion-formation rule fails to introduce the recursive call $gcd(x\ y-x)$ because we cannot establish its termination condition; indeed, if we did introduce this recursive call, the program would certainly not terminate. Instead, we look for some alternate means of satisfying Goal 6,

**compute** $max\{z : z|x \text{ and } z|y-x\}$ ,

which, since $x = 0$, is reduced to

**Goal 11:** **compute** $max\{z : z|0 \text{ and } z|y\}$ .

By application of the three rules

$u|v \Rightarrow true$ if $v = 0$,

$true \text{ and } P \Rightarrow P$ , and

$max\{u : u|v\} \Rightarrow v$ if $v$ is a positive integer

in succession, we obtain

**Goal 12:** **compute** $y$ .

The last rule could be applied because in this case $x = 0$, and thus $y \neq 0$ (since $x \neq 0$ or $y \neq 0$), and $y > 0$ (since $y$ is nonnegative).

Now $y$ is a primitive program segment that solves our problem in this final case. The complete $gcd$ program is

```
gcd(x y) <== if y < x
             then gcd(y x)
             else if x ≠ 0
                  then gcd(x y−x)
                  else y .
```

This is a version of the "subtractive" *gcd* algorithm.

> Well-founded orderings were first invoked to prove properties of recursive programs by Burstall [1969]. The theorem-proving system of Boyer and Moore [1977] also constructs lexicographic orderings.

The particular program we obtain depends on the transformation rules we have at our disposal and the choices we make during the derivation process. For example, if we had the additional rules

$gcd(u\ v) => 2\cdot gcd(u/2\ v/2)$ if $u$ and $v$ are even,

$gcd(u\ v) => gcd(u/2\ v)$ if $u$ is even and $v$ is odd, and

$gcd(u\ v) => gcd(u\ v/2)$ if $u$ is odd and $v$ is even,

we could have obtained the "binary" *gcd* program

```
gcd(x y) <== if even(x)
             then if even(y)
                  then 2·gcd(x/2 y/2)
                  else gcd(x/2 y)
             else if even(y)
                  then gcd(x y/2)
                  else if y < x
                       then gcd(y x)
                       else if x ≠ 0
                            then gcd(x y−x)
                            else y .
```

This program turns out to be quite efficient for implementation on a binary machine, in which division and multiplication by two can be represented as right and left shifts, respectively (or vice versa, depending on which side of the machine we are standing on). Of course, nothing in the technique guarantees that an efficient program will be derived.

## D. Strategic Controls

Up to now we have developed programs by applying transformation rules to goals without considering how to select the rule to be applied; the proper rule seemed to appear by magic when it was relevant. If we have hundreds of rules at our disposal, how do we retrieve the applicable ones? Of the many rules that can be applied in a given situation, not all will lead to a primitive program. If more than one rule applies to a goal, how do we decide which to attempt?

If the program is being developed by hand, we can rely on the programmer's knowledge and intuition. However, if we expect this process to be performed by an automatic synthesis system, the basis for our strategic decisions must be made explicit. In this section, we will discuss some strategic methods for directing the transformation rules.

The strategic controls that we have incorporated into our own program-synthesis system may be outlined as follows. When a goal is proposed, the rules that seem applicable are selected by *pattern-directed invocation.* Of all the selected rules, one is chosen according to a given *rule ordering*; this rule is attempted first. Each rule may be provided with a number of *strategic conditions*, which prevent it from being applied foolishly. If the strategic conditions are not satisfied, or if the rule does apply but does not lead to a primitive program, we *backtrack* and consider the next applicable rule chosen by the rule ordering. Let us discuss each of these methods in more detail.

- *Pattern-directed invocation*: The rules are indexed by the patterns to which they can be applied. For example, the *all* decomposition rule

$$P(all(l)) \Rightarrow P(head(l)) \text{ and } P(all(tail(l))) \quad \text{if } l \text{ is a nonempty list}$$

is classified according to its left-hand side, $P(all(l))$. When a new goal is proposed, all those rules whose patterns match the goal are retrieved. Thus, the above rule and the vacuous rule

$$P(all(l)) \Rightarrow true \quad \text{if } l \text{ is the empty list,}$$

would both be invoked when the goal **compute** $x < all(l)$ is proposed. This method of retrieving a rule when it seems applicable is termed *pattern-directed invocation.*

- *Rule ordering*: It often happens that more than one transformation rule will match the same goal. However, sometimes we can decide a priori that one rule should be attempted before another. For example, if the vacuous rule

$$P(all(l)) \Rightarrow true \quad \text{if } l \text{ is the empty list}$$

and the recursion-formation rule both match the same goal, the vacuous rule should always be attempted first; the recursion-formation rule imposes the input and termination conditions, which may be time-consuming to verify. Furthermore, if both rules do apply, the program segment *true* is preferable to a recursive call.

On the other hand, if the decomposition rule

$$P(\mathit{all}(l)) \Rightarrow P(\mathit{head}(l)) \text{ and } P(\mathit{all}(\mathit{tail}(l))) \quad \text{if } l \text{ is a nonempty list}$$

and the recursion-formation rule both match the same goal, we prefer to attempt the recursion-formation rule first; the decomposition rule produces a nonprimitive subgoal more complex than the original goal, while the recursion-formation rule is guaranteed to produce a primitive recursive call.

- *Strategic conditions*: We have seen that a transformation rule may impose logical conditions, which must be satisfied to ensure a valid application of the rule. By the same token, a rule may have strategic conditions, which prevents it from being applied foolishly. For example, in introducing a conditional expression *if* $P$ *then* $s_1$ *else* $s_2$ or the recursive call $f(t)$ we imposed the strategic condition that the condition $P$ or the argument $t$ be primitive; this was to ensure that the resulting expression would itself be primitive.

Two more examples: if we introduce the logical rule

$$P \text{ and } Q \Rightarrow Q \text{ and } P,$$

or the integer rule

$$u|v \text{ and } u|w \Rightarrow u|v \text{ and } u|w-v\ ,$$

we must give them each strategic conditions to ensure that they are not applied repeatedly to the subexpressions that they themselves produce; otherwise, we may obtain an endless sequence, e.g.

$$P \text{ and } Q\ ,\quad Q \text{ and } P\ ,\quad P \text{ and } Q\ ,\quad \ldots\ .$$

Good strategic conditions improve the general performance of a system, but they may prevent it from finding some trickier, less intuitive solutions.

- *Backtracking*: If applying one rule to a goal fails to lead to a primitive program segment, the system will *backtrack*, and attempt to apply other applicable rules to the same goal.

For instance, in constructing the *gcd* program, we applied the rule

$$P \text{ and } Q \Rightarrow Q \text{ and } P$$

to Goal 1,

**compute** $max\{z : z|x \text{ and } z|y\}$ ,

to form Goal 2,

**compute** $max\{z : z|y \text{ and } z|x\}$ .

In the case in which $x \le y$, we failed to derive a primitive program segment from Goal 2; therefore, we backtracked and considered other rules that matched Goal 1. As it turned out, the rule

$u|v \text{ and } u|w \Rightarrow u|v \text{ and } u|w-v$

applied to Goal 1 to yield Goal 6,

**compute** $max\{z : z|x \text{ and } z|y-x\}$ .

In addition to these general strategic methods for controlling transformation rules, there are special strategic techniques associated with particular rules. One of these techniques is the subject of the next subsection.

Pattern-directed invocation was introduced as a feature of the PLANNER programming language for artificial-intelligence research (Hewitt [1971]).

### The Redundant-Test Strategy

The conditional-formation rule will introduce a case analysis when we fail to prove or disprove a condition $P$. We consider separately the case in which $P$ is true and the case in which $P$ is false, construct program segments $s_1$ and $s_2$ to handle each case, and combine these segments into the conditional expression

$\textit{if } P \textit{ then } s_1 \textit{ else } s_2$ .

However, it is possible that one of these segments, say $s_2$, does not depend on the corresponding case assumption, that $P$ is false. In this situation, the segment $s_2$ itself will solve our problem regardless of whether $P$ is true or false; constructing the other segment $s_1$ would be a waste of effort.

The *redundant-test strategy* prevents such irrelevant conditional expressions from being formed. According to this strategy, in introducing a case analysis we always consider first the negative case, in which $P$ is false. If we then succeed in constructing a program segment $s_2$ that solves our problem without ever using the case assumption that $P$ is false, then this segment solves the entire problem. We do not consider the positive case, in which $P$ is true, and we do not generate a conditional expression.

We always consider the negative case first because in the positive case, the assumption that $P$ is true will always be used by the rule that imposed the condition; therefore, we can never escape considering the negative case.

For example, suppose, in constructing the *gcd* program, we are given the *rem* rule

$$u|v \textit{ and } u|w \Rightarrow u|v \textit{ and } u|rem(w\ v) \qquad \text{if } v \neq 0$$

instead of the *minus* rule

$$u|v \textit{ and } u|w \Rightarrow u|v \textit{ and } u|w-v\ ,$$

where $u$, $v$, and $w$ are nonnegative integers. (The *rem* rule states that the common divisors of $v$ and $w$ are the same as the common divisors of $v$ and $rem(w\ v)$.) Recall that in developing our previous *gcd* program, we introduced a case analysis on the condition $y < x$ in an attempt to introduce a recursive call $gcd(y\ x)$. Now, according to the redundant-test strategy, we will first consider the negative case, in which $x \leq y$. In this case we will apply the *rem* rule and eventually develop the program segment

```
if x ≠ 0
then gcd(rem(y x) x)
else y
```

without ever using the case assumption that $x \leq y$. Consequently, we need never consider the positive case, in which $y < x$. The above segment solves the entire problem, so our final program is simply

```
gcd(x y) <== if x ≠ 0
             then gcd(rem(y x) x)
             else y .
```

This is a version of the Euclidean *gcd* algorithm.

In describing a program derivation in which a case analysis is introduced and later eliminated by the redundant-test strategy, we will often omit mentioning the case analysis altogether. For example, in developing either of the above *gcd* programs, we introduce a case analysis on the condition $y = 0$ as well as on the condition $x = 0$; this case analysis on $y = 0$ is eliminated by the redundant test strategy, and never appears in our discussion.

# 3. EXTENSIONS OF RECURSION FORMATION

## A. Generalization

Recursive calls have been introduced when a new subgoal is discovered to be a precise instance of the top-level goal. But what if the subgoal is an instance not of the top-level goal but of a somewhat more general expression? In such cases, it may be advisable to construct a new procedure (or subroutine) to compute the more general expression, and to achieve our original goal by a call to the new procedure. Although the new procedure attempts to solve a more general problem, that problem may nevertheless be easier to solve.

Generalization is already commonplace in the theorem-proving context: paradoxically, it is often necessary, in proving a theorem by mathematical induction, to prove a more general theorem, so that the induction hypothesis will be strong enough to prove the inductive step. In program synthesis, induction is analogous to recursion: we attempt to construct a program to compute a more general goal so that the recursive call will be strong enough to achieve the desired subgoal.

As before, we will explain the method in the context of an example. We will not follow the precise order dictated by the strategic controls in constructing the program. Because we have considered a similar program, *lessall*(*x l*), previously, we will be a bit more brief in our exposition.

Suppose we want to construct a program *headtail*(*l*) to test whether the head of a nonempty list *l* is less than every element of its tail. The specifications for this program may be expressed as

*headtail*(*l*) <== **compute** *head*(*l*) < *all*(*tail*(*l*))
**where** *l* is a nonempty list of numbers.

Our top-level goal is then

**Goal 1:** **compute** *head*(*l*) < *all*(*tail*(*l*)) .

Recall that we have introduced two rules that explicate the *all* construct: the vacuous rule

*P*(*all*(*l*)) => *true* if *l* is the empty list,

and the decomposition rule

*P*(*all*(*l*)) => *P*(*head*(*l*)) *and P*(*all*(*tail*(*l*)) if *l* is a nonempty list.

These rules, together with the conditional-formation rule, account for the introduction of a case analysis into our derivation, and the subsequent formation of a conditional expression in our final program. In the case that *tail(l)* is empty, the vacuous rule reduces the goal to the primitive segment *true*; in the other case, in which *tail(l)* is not empty, the decomposition rule reduces the goal to computing the conjunction of two expressions:

**Goal 2:** **compute** $head(l) < head(tail(l))$

and

**Goal 3:** **compute** $head(l) < all(tail(tail(l)))$ .

Goal 2 is already a primitive expression. We have yet to consider Goal 3; however, the program constructed so far is

```
headtail(l) <== if empty(tail(l))
                   then true
                   else head(l) < head(tail(l)) and
                        ...
```

An attempt to satisfy Goal 3 by the recursion-formation rule fails, because Goal 3 is not a precise instance of Goal 1,

**compute** $head(l) < all(tail(l))$ ;

the $l$ on the left-hand side of Goal 1 corresponds to $l$ in the subgoal, but the $l$ on the right-hand side corresponds to *tail(l)*. However, Goal 3 is an instance of a more general goal,

**Goal 1 (generalized):** **compute** $head(l_1) < all(tail(l_2))$ ,

obtained from Goal 1 by introducing new variables $l_1$ and $l_2$ in place of the left- and right-hand occurrences of $l$, respectively. This suggests that we attempt to construct a procedure $headtailgen(l_1\ l_2)$ to achieve the generalized Goal 1 instead of the original version. Thus, the output specification for the new procedure will be

$headtailgen(l_1\ l_2)$ <== **compute** $head(l_1) < all(tail(l_2))$ .

This procedure will test whether the head of $l_1$ is less than every element of the tail of $l_2$, where $l_1$ and $l_2$ may be distinct lists.

We can now set aside our original derivation, and satisfy the original Goal 1 by a call to the more general procedure instead; the resulting *headtail* program will be simply

*headtail(l)* <== *headtailgen(l l)* .

It remains to construct the more general procedure *headtailgen*, i.e. to achieve the generalized Goal 1. The derivation of the generalized goal will attempt to mirror the original derivation; our hope is that this time the top-level goal is general enough so that the previous obstacle encountered in introducing the recursive call will be overcome.

> In general, suppose we are developing a program whose specifications are of form
>
> $f(x)$ <== **compute** $P(a(x))$
> **where** $Q(x)$ .
>
> Then our top-level goal is of form
>
> **Goal A:** **compute** $P(a(x))$ .
>
> Suppose that in developing the program we encounter a subgoal
>
> **Goal B:** **compute** $P(b(x))$
>
> that is not an instance of Goal A, but that is an instance of the more general expression
>
> **compute** $P(y)$.
>
> Then the *generalization rule* proposes that we attempt to construct a new procedure whose output specification is
>
> $g(y)$ <== **compute** $P(y)$ .
>
> We can thus satisfy the original Goal A by a call to the new procedure; the resulting program $f$ will be
>
> $f(x)$ <== $g(a(x))$ .
>
> To ensure that the calls to the new procedure $g$ will be primitive, we do not apply the generalization rule unless $a(x)$ and $b(x)$ are primitive.

> The top-level goal of the new derivation will be the generalized Goal A, **compute** $P(y)$. We will attempt to mirror the steps of the original derivation; that is, we try to apply to the new goal the same rules that we applied earlier to the original Goal A in deriving the original Goal B. Our hope is that the goal in the new derivation corresponding to the original

Goal B will turn out to be an instance of the generalized Goal A, and that it will be achieved by a recursive call to $g$. However, there is no guarantee that the same sequence of rules will be applicable to the generalized Goal A, or that if we succeed in deriving a generalized Goal B, it will turn out to be an instance of the generalized Goal A. If the derivation fails for either reason, we abandon the generalization and look for other ways to achieve the original Goal B. (This is a very conservative strategy; a more adventurous approach would be to try to use as much as possible of the original derivation, but to seek other ways of progressing when the original derivation fails.)

We have postponed describing the input specification for the new procedure $g$. It is to our advantage to have as few conditions in this specification as possible, because we must check each of these conditions every time a procedure call to $g$ is introduced. For this reason, rather than attempting to formulate the new input specification in advance, we prefer to proceed with the derivation of $g$ and add to the input specification only those conditions that are needed to complete the derivation. In other words, we form the input specification for $g$ *incrementally*.

Thus, if in the course of the derivation we fail to prove a desired condition $S(y)$, we consider adding this condition to the input specification of $g$. However, every time a call $g(u)$ to the procedure $g$ has been introduced previously in the synthesis, we must go back and check that the additional input condition $S(u)$ is satisfied. In particular, because the main program

$$f(x) \Leftarrow\!\!= g(a(x))$$

contains a procedure call $g(a(x))$, we must check that condition $S(a(x))$ is satisfied.

Often, conditions are added to the input specification simply to ensure that the output specification is meaningful.

Returning to our example, we attempt to construct the more general procedure $headtailgen(l_1\ l_2)$ that achieves the generalized Goal 1,

**compute** $head(l_1) < all(tail(l_2))$ .

However, this goal is not meaningful unless

$l_1$ and $l_2$ are nonempty lists.

We cannot prove this condition about our arbitrary inputs $l_1$ and $l_2$; therefore, we must add it to the input specification for the new procedure. Because the main program *headtail*($l$) contains the call *headtailgen*($l$ $l$), we first check that the arguments $l$ and $l$ for the call satisfy the proposed condition. Thus, we have to show that

$l$ and $l$ are nonempty lists,

i.e.,

$l$ is a nonempty list.

But this is exactly the input specification for the main program.

We attempt to apply to the generalized Goal 1 the same sequence of rules that we applied to the original Goal 1 earlier. Applying the vacuous rule in the case where *tail*($l_2$) is empty, we derive the primitive program segment *true*; applying the decomposition rule in the case where *tail*($l_2$) is not empty, we decompose the generalized Goal 1 into computing the conjunction of two expressions:

**Goal 2 (generalized):** **compute** *head*($l_1$) < *head*(*tail*($l_2$))

and

**Goal 3 (generalized):** **compute** *head*($l_1$) < *all*(*tail*(*tail*($l_2$))) .

The new Goal 2 is a primitive expression as before; however, this time the new Goal 3 is a precise instance of the generalized Goal 1

**compute** *head*($l_1$) < *all*(*tail*($l_2$)) ;

therefore, the recursion-formation rule proposes that we achieve the generalized Goal 3 by a recursive call *headtailgen*($l_1$ *tail*($l_2$)) to the new procedure. The arguments $l_1$ and *tail*($l_2$) can be shown in this case to satisfy the input condition that

$l_1$ and *tail*($l_2$) are nonempty lists,

because $l_1$ and $l_2$ are nonempty lists (the new input condition) and *tail*($l_2$) is not empty (the case assumption). The termination condition is established because the second argument *tail*($l_2$) of the recursive call is a sublist of the second input $l_2$.

The complete final program is then

*headtail*($l$) <== *headtailgen*($l$ $l$)

where

```
headtailgen($l_1$ $l_2$) <== if empty(tail($l_2$))
                              then true
                              else head($l_1$) < head(tail($l_2$)) and
                                   headtailgen($l_1$ tail($l_2$)) .
```

When it is successful, the generalization principle results in the construction of a stronger program than originally required. If the new specifications are too general, however, the corresponding program can actually be more difficult to construct than the original. For this reason, we must impose conservative strategic controls on the application of the generalization principle. For all the examples in this paper, the only generalizations required involve replacing a constant by a variable, or one occurrence of a variable by a new variable; in general, it is necessary to replace more complex terms by variables.

> For examples of theorem-proving systems that generalize the theorems they are about to prove by induction, see Boyer and Moore [1975], Brotz [1973], and Aubin [1975] . Siklossy [1974] proposed applying this technique to program synthesis.

## B. The Formation of Subsidiary Procedures

We form a recursive call when a subgoal is discovered to be an instance of the top-level goal. But what if the subgoal is an instance, not of the top-level goal, but of some other subgoal? In this section, we show how such a situation can lead to the formation of *subsidiary procedures* (or subroutines) .

As before, we will consider the general case in the context of a specific example. The program to be constructed, *allall(l m)*, is intended to test whether every member of a given list *l* of numbers is less than every member of another such list *m*. The specifications can be expressed as

*allall(l m)* <== **compute** *all(l)* < *all(m)* ,
**where** *l* **and** *m* **are lists of numbers.**

The top-level goal is thus

**Goal 1:** **compute** *all(l)* < *all(m)* .

As before, we will employ the vacuous rule

$P(all(l)) \Rightarrow true$ if $l$ is the empty list

and the decomposition rule

$P(all(l)) \Rightarrow P(head(l))$ *and* $P(all(tail(l)))$ if $l$ is a nonempty list.

In the case in which $l$ is empty, the vacuous rule reduces Goal 1 to the primitive program segment *true*; in the other case, the decomposition rule reduces the goal to computing the conjunction of two expressions:

**Goal 2:** **compute** $head(l) < all(m)$

and

**Goal 3:** **compute** $all(tail(l)) < all(m)$ .

Goal 3 is discovered to be an instance of the top-level goal, with the inputs $l$ and $m$ replaced by $tail(l)$ and $m$. Therefore, the recursion-formation rule replaces this goal by a recursive call $lessall(tail(l)\ m)$; the input condition is easily checked, and the termination condition is proved because $tail(l)$ is a proper sublist of $l$.

We have yet to consider Goal 2; the program constructed so far has the form

```
allall(l m) <== if empty(l)
                then true
                else ... and
                     allall(tail(l) m) .
```

Goal 2, **compute** $head(l) < all(m)$, is decomposed in a manner similar to Goal 1. In the case where $m$ is empty, the vacuous rule transforms this expression to the primitive program segment *true*. In the other case, the decomposition rule reduces this goal to computing the conjunction of two expressions:

**Goal 4:** **compute** $head(l) < head(m)$

and

**Goal 5:** **compute** $head(l) < all(tail(m))$ .

Goal 4 is a primitive expression that can be computed directly. Goal 5 is an instance not of the top-level goal but of the intermediate Goal 2, **compute** $head(l) < all(m)$, with the inputs $l$ and $m$ replaced by $l$ and $tail(m)$. This suggests that we might achieve Goal 5 by a recursive call not to the entire program *allall* but to the segment of *allall* that achieves Goal 2. For this purpose, we must introduce a subsidiary procedure $headall(l\ m)$ corresponding to this segment. Thus, the output specification for the new procedure will be

$headall(l\ m)$ <== **compute** $head(l) < all(m)$ .

(This procedure tests whether the head of $l$ is less than every element of $m$.) Then we can achieve Goal 5,

**compute** $head(l) < all(tail(m))$,

by a recursive call $headall(l\ tail(m))$ to the new procedure.

In general, suppose we are developing a program whose specifications are of the form

$f(x)$ <== **compute** $P(x)$
**where** $Q(x)$ ,

and we encounter a subgoal

**Goal B:** **compute** $R(t)$ ,

which is an instance of some previously generated subgoal

**Goal A:** **compute** $R(x)$ .

We assume that Goal A is some ancestor of Goal B other than the top-level goal. The *procedure-formation rule* proposes that we introduce a new procedure $g$ whose output description is

$g(x)$ <== **compute** $R(x)$ ,

so that we can achieve Goal B by a recursive call $g(t)$. Then we set aside the original derivation for Goal A, and achieve the goal by a call $g(x)$ to the new procedure.

As in the previous section, we prefer to formulate the input specifications for the new procedure $g$ incrementally, rather than attempting to express this specification in advance. Again, it is to our advantage to have as few conditions as possible in the input specification for $g$, because each of these conditions must be checked every time a call to $g$ is introduced. We add to the new input specification only those conditions that are needed in the course of the derivation of $g$.

Thus, if in constructing the procedure $g$ we fail to prove some condition

> $S(x)$, we consider adding this condition to the input specification for $g$. However, every time a call $g(u)$ to the new procedure has been introduced earlier in the synthesis, we must go back and check that the additional input condition $S(u)$ is satisfied. In particular, because the main program $f$ now contains a call $g(x)$ to achieve Goal A, we must check that $S(x)$ holds when this call is executed. For this purpose, we may use the input specifications for $f$ or any of the case assumptions that occur in the derivation of Goal A.
>
> Goal A, **compute** $R(x)$, now becomes the top-level goal in the construction of the procedure $g$. Initially, we mirror the steps of the original derivation; that is, we apply in the new derivation the same sequence of steps that we applied originally, adding conditions to the input specification of $g$ as necessary. Goal B, **compute** $R(t)$, will again be introduced, and will again be an instance of Goal A, **compute** $R(x)$. This time, however, Goal A is the top-level goal, so the recursion-formation rule can be applied to satisfy Goal B with a recursive call $g(t)$, provided that the input and termination conditions are satisfied. This input condition for such a recursive call is the same as usual; however, the termination condition is more complex, and will not be discussed until Section 3D.
>
> We may need to achieve other goals to complete the derivation of the main procedure $f$ and the subsidiary procedure $g$. Of course, in continuing these derivations we may introduce still more subsidiary procedures.

Returning to our *allall* example, recall that we developed a subgoal

**compute** *head*(*l*) < *all*(*tail*(*m*))

(Goal 5), which we observed to be an instance of its ancestor subgoal

**compute** *head*(*l*) < *all*(*m*)

(Goal 2). Therefore, the procedure-formation rule suggests introducing a new procedure, *headall*, whose output specification is

*headall*(*l m*) <== **compute** *head*(*l*) < *all*(*m*) .

The partial program description derived from Goal 2 is set aside; this goal is now satisfied by a call *headall*(*l m*) to the new procedure. Thus, the final *allall* program is

```
allall(l m) <== if empty(l)
                  then true
```

```
                        else  headall(l m) and
                                allall(tail(l) m) .
```

We have yet to complete the construction of the subsidiary procedure *headall*. The top-level goal for the procedure is Goal 2,

**compute** $head(l) < all(m)$ .

This expression is not well-formed unless

$l$ and $m$ are lists  
and $l$ is not empty.

By our incremental specification technique, we consider adding these conditions to the input specification for *headall*. Because a call *headall*($l$ $m$) has already been introduced in the main program to achieve Goal 2, we must check that these conditions are satisfied when this call is made. However, the first condition is the input specification for the main program, and the second condition holds because Goal 2 was introduced under the assumption that $l$ is not empty. Therefore, these conditions may safely be added to the input specification for *headall*.

To complete the derivation of the *headall* procedure, we begin by mirroring the derivation leading from Goal 2 in the original synthesis. We again introduce Goals 4 and 5. Goal 5,

**compute** $head(l) < all(tail(m))$ ,

is again an instance of Goal 2,

**compute** $head(l) < all(m)$ .

However, this time Goal 2 is the top-level goal, and the recursion-formation rule can now introduce the recursive call *headall*($l$ *tail*($m$)). (The input and termination conditions for this call are straightforward.) The complete program we derive is thus

```
allall(l m) <== if empty(l)
                then true
                else headall(l m) and
                       allall(tail(l) m) ,
```

where

```
headall(l m) <== if empty(m)
                   then true
                   else head(l) < head(m) and
                          headall(l tail(m)) .
```

### Another Example

Using the same basic principles as in the *lessall* example, but employing some additional rules for the set-theoretic domain, we can construct a program to compute the Cartesian product $cart(s\ t)$ of two sets $s$ and $t$. The specifications for this program are

$$\begin{aligned} &cart(s\ t) \Leftarrow\!\!=\; \textbf{compute}\ \{\ (x\ y) : x \in s\ \textit{and}\ y \in t\ \} \\ &\qquad \textbf{where}\ s \text{ and } t \text{ are sets.} \end{aligned}$$

The rules for sets employed in this synthesis are the *empty-set-formation* rule,

$$\{u : \textit{false}\} \Rightarrow \{\ \}$$

(where { } is the empty set), the *union-formation* rule

$$\{u : P(u)\ \textit{or}\ Q(u)\} \Rightarrow \{u : P(u)\} \cup \{u : Q(u)\}$$

(where ∪ denotes the union of two sets), the *equality-elimination* rule

$$\{u : u = t\} \Rightarrow \{t\}$$

(where $u$ and $t$ are expressions with no variables in common), and the definition of the *member* relation $\in$ . We assume that the empty set { }, the functions $head(s)$ and $tail(s)$, the union function ∪, and the notations for the singleton set {s} and the pair $(s\ t)$ are among the primitives of our target language.

We will be very brief. In deriving the program from the specifications, we decompose the output specification into the expression

$$\begin{aligned} &\{\ (x\ y) : x = head(s)\ \textit{and}\ y \in t\ \} \cup \\ &\{\ (x\ y) : x \in tail(s)\ \textit{and}\ y \in t\ \}\ , \end{aligned}$$

corresponding to the case in which $s$ is nonempty. The second subexpression,

$$\{\ (x\ y) : x \in tail(s)\ \textit{and}\ y \in t\ \}\ ,$$

can be computed by a simple recursive call $cart(tail(s)\ t)$.

It remains to compute the first subexpression, i.e.,

**Goal A:** **compute** $\{\ (x\ y) : x = head(s)\ \textit{and}\ y \in t\ \}$ .

This expression decomposes further, yielding

$$\{\ (x\ y) : x = head(s)\ and\ y = head(t)\ \} \cup$$
$$\{\ (x\ y) : x = head(s)\ and\ y \in tail(t)\ \}$$

in the case in which $t$ is nonempty. The first subexpression,

$$\{\ (x\ y) : x = head(s)\ and\ y = head(t)\ \}\ ,$$

reduces directly to the primitive expression

$$\{\ (head(s)\ head(t))\ \}\ .$$

It remains to compute the second subexpression, i.e.,

**Goal B:** **compute** $\{\ (x\ y) : x = head(s)\ and\ y \in tail(t)\ \}$ .

Goal B is an instance of Goal A; therefore, we introduce a new procedure *carthead*, whose output specification is

*carthead*(*s t*) <== **compute** $\{\ (x\ y) : x = head(s)\ and\ y \in t\ \}$ .

(This procedure computes the Cartesian product of the singleton set $\{head(s)\}$ and $t$.) To ensure that this specification is well-formed, we are forced to introduce the condition

> $s$ and $t$ are sets
> and $s$ is not empty

as the input specification for the subsidiary procedure.

Then Goal A is satisfied by a call *carthead*(*s t*) to the new procedure, while Goal B is satisfied by a recursive call *carthead*(*s tail*(*t*)). The complete Cartesian product program is

```
cart(s t) <==  if empty(s)
               then { }
               else  carthead(s t) ∪
                        cart(tail(s) t)) ,
```

where

```
carthead(s t) <== if empty(t)
                  then { }
                  else  {(head(s) head(t))} ∪
                          carthead(s tail(t)) .
```

The Cartesian-product example is derived from Darlington [1975].

## C. The Generalization of Subsidiary Procedures

In our discussion of subsidiary-procedure formation, we introduced a procedure only if a subgoal (Goal B) is discovered to be a precise instance of a previously generated subgoal (Goal A). We further required that Goal A be a direct ancestor of Goal B (other than the top-level goal). However, what if Goal A is not actually an ancestor of Goal B but occurs somewhere else in the synthesis? Or what if Goal B is not a precise instance of Goal A, but of a somewhat more general expression? In fact, the techniques we have already introduced extend naturally to this more general situation, as we will see in our next example. This example will also serve to illustrate how program-synthesis techniques can be applied to transform an already-constructed program.

Suppose we are given the following program *reverse*($l$) for reversing the elements of a list $l$ :

```
reverse(l) <== if empty(l)
               then nil
               else append(reverse(tail(l))
                           list(head(l))) ,
```

where *nil* is the empty list and *append*($l_1$ $l_2$) is the program for appending the elements of two lists, given by

```
append(l_1 l_2) <== if empty(l_1)
                    then l_2
                    else cons(head(l_1)
                              append(tail(l_1) l_2)) .
```

This *reverse* program is not very efficient because its execution may involve many calls to *append*; moreover, each time *append* is called it makes a new copy of its first argument.

Let us consider the given *reverse* program to be the specification for another *reverse* program. Even though we have a program to compute the *append* function, let us treat *append* as a nonprimitive construct. Thus, we will be forced to transform our given program into an equivalent program that does not use *append*. Our hope is that the resulting program will be more efficient.

We assume that we have the following rules that explicate the *append* construct:

$$append(l_1\ l_2) \Rightarrow l_2 \quad \text{if } l_1 \text{ is the empty list}$$

$$append(l_1\ l_2) \Rightarrow cons(head(l_1)\ append(tail(l_1)\ l_2)) \quad \text{if } l_2 \text{ is a nonempty list,}$$

and

$$append(append(l_1\ l_2)\ l_3) \Rightarrow append(l_1\ append(l_2\ l_3))\ .$$

These rules are derived from the *append* program itself. In addition, we will use the given *reverse* program as a transformation rule:

```
reverse(l) => if empty(l)
              then nil
              else append(reverse(tail(l))
                              list(head(l))) .
```

We will also apply several rules based on the properties of list structures.

Our top-level goal is

```
Goal 1:  compute if empty(l)
                 then nil
                 else append(reverse(tail(l))
                                 list(head(l))) .
```

The "nonprimitive" construct *append* appears in the *else* branch of the goal. Applying the transformation rules

*list*($y_1$ $y_2$ … $y_n$) => *cons*($y_1$ *list*($y_2$ … $y_n$))   if $n \geq 1$

and

*list*( ) => *nil*

to the *else* clause, we obtain

Applying to the subexpression *reverse*(*tail*(*l*)) the rule for *reverse*, and "pulling out" the conditional expression using the rule

*f*(*if P then* $s_1$ *else* $s_2$) => *if P then* *f*($s_1$) *else* *f*($s_2$) ,

we obtain

Applying the rule

*append*($l_1$ $l_2$) => $l_2$   if $l_1$ is the empty list

to the *then* clause, and applying the rule

*append*(*append*($l_1$ $l_2$) $l_3$) => *append*($l_1$ *append*($l_2$ $l_3$))

to the *else* clause, we obtain

**Goal 4: compute** *if empty(tail(l))*
*then cons(head(l) nil)*
*else append(reverse(tail(tail(l)))*
*append(list(head(tail(l)))*
*cons(head(l) nil)))* .

Let us focus our attention on the *else* branch of this goal.

**Goal 5: compute** *append(reverse(tail(tail(l)))*
*append(list(head(tail(l)))*
*cons(head(l) nil)))* .

By the rules for *list*, *append*, and *cons*, this reduces to

**Goal 6: compute** *append(reverse(tail(tail(l)))*
*cons(head(tail(l))*
*cons(head(l) nil)))* .

This goal is not a precise instance of the higher-level Goal 2,

**compute** *append(reverse(tail(l))*
*cons(head(l)*
*nil))* ,

because the expression *cons(head(l) nil)* in Goal 6 coincides with the constant *nil* in Goal 2. However, Goal 6 is a precise instance of the somewhat more general expression

**compute** *append(reverse(tail(l))*
*cons(head(l)*
*m))* ,

obtained from Goal 2 by replacing the constant *nil* by a new variable *m*.

We have developed a situation in which a subgoal is a precise instance, not of the previously generated subgoal, but of a somewhat more general expression. In other words, we have found that

**Goal B: compute** $R(b(x))$

and the previously generated

**Goal A: compute** $R(a(x))$

are both instances of the more general expression

**compute** $R(y)$ .

(Note that we do not need to assume that Goal A is actually an ancestor of Goal B, or even that both appear in the synthesis of the same procedure.)

In this situation, the *extended procedure-formation rule* proposes introducing a new subsidiary procedure $g(y)$ whose purpose is to achieve both goals. The output specification for $g$ will be

$g(y)$ <== **compute** $R(y)$ .

We intend to achieve Goal A, **compute** $R(a(x))$, by a call $g(a(x))$, and to achieve Goal B, **compute** $R(b(x))$, by a call $g(b(x))$. (In the special case where Goal A is already the top-level goal of some procedure that achieves it, and Goal B is a precise instance of Goal A, there is of course no need to introduce a new procedure to achieve Goal A.)

The input specification for the new procedure $g$ is formed incrementally as before. The top-level goal in the derivation of $g$ is

**Goal A (generalized): compute** $R(y)$ .

In constructing the subsidiary program $g$, we begin by attempting to mirror the original derivation leading from Goal A, adding conditions to the input specification as necessary. All the techniques presented previously can then be applied to complete the derivation of $g$.

Returning to our example, recall that the extended procedure-formation rule proposed introducing a new subsidiary procedure *reversegen(l m)* to compute the more general expression. Thus, the output specification for *reversegen* is

```
reversegen(l m) <==  compute append(reverse(tail(l))
                                    cons(head(l)
                                         m)) .
```

(Intuitively, the *reversegen(l m)* reverses a nonempty list *l* and appends the result to *m* .)

Now, Goal 2 in the derivation of the main program is achieved by a call *reversegen(l nil)* to the subsidiary procedure. The final *reverse* program is then

```
reverse(l) <==  if empty(l)
                then nil
```

```
                    else reversegen(l nil) .
```

It remains to complete the derivation of *reversegen*. The top-level goal for this derivation is obtained directly from the output specification:

```
Goal 2 (generalized):  compute append(reverse(tail(l))
                                      cons(head(l)
                                           m)) .
```

To ensure that this expression is well-formed, we add the conditions

*l* and *m* are lists
and *l* is nonempty

incrementally to the input specification for the *reversegen* procedure. We then attempt to mirror the original derivation leading from Goal 2. We succeed in applying the same rules as before, ultimately obtaining

```
Goal 6 (generalized):  compute append(reverse(tail(tail(l)))
                                      cons(head(tail(l))
                                           cons(head(l)
                                                m))) .
```

This time, the generalized Goal 6 is indeed an instance of the generalized Goal 2, obtained by replacing *l* with *tail*(*l*) and *m* with *cons*(*head*(*l*) *m*). Therefore, we can achieve the new Goal 6 by a recursive call *reversegen*(*tail*(*l*) *cons*(*head*(*l*) *m*)) to the subsidiary procedure. The final *reverse* program we obtain is thus

```
reverse(l) <==  if empty(l)
                then nil
                else reversegen(l nil)
```

where

```
reversegen(l m) <==  if empty(tail(l))
                     then cons(head(l) m)
                     else reversegen(tail(l)
                                     cons(head(l) m)) .
```

This is a better *reverse* program than the one we were originally given. Not only has the expensive *append* program been eliminated, but by good fortune the new procedure *reversegen* we have obtained is of a special form, for which the recursion can be implemented efficiently without the use of a stack.

The *reverse* example follows Burstall and Darlington [1977]. Their system does not perform the generalization automatically.

## D. Systems of Mutually Recursive Procedures

In the above examples we have used the usual techniques for showing the termination of the programs and procedures we construct. However, certain situations arise in introducing subsidiary procedures that require this technique to be strengthened. In particular, we can form systems of *mutually recursive procedures*, i.e. procedures each of which may contain calls to the others. Let us see how such a system can emerge.

Suppose that one subgoal in the derivation of a subsidiary procedure $g$ is achieved by a call to the main program $f$. Then the program $f$ will be expressed in terms of a call to the procedure $g$,

$$f(x) \Leftarrow\!= \ldots g(u) \ldots ,$$

while $g$ will be expressed in terms of a call to the main program $f$,

$$g(y) \Leftarrow\!= \ldots f(v) \ldots .$$

Such a system of mutually recursive procedures can fail to terminate, say if $f$ calls $g$, $g$ calls $f$, $f$ calls $g$ again, and so on indefinitely. The naive approach for showing the termination of such a system is to show that all the inputs and arguments belong to some well-founded set $W$, and that

$$u \prec x \text{ and } v \prec y$$

under the ordering $\prec$ of $W$. However, there are systems whose termination cannot be shown by this approach; for example, if $u$ is $x$ itself, then no well-founded ordering will allow us to show $u \prec x$. Furthermore, in some systems, $f$ and $g$ may apply to different domains; $f$ may apply to lists, say, and $g$ may apply to numbers; in such a case, it may be difficult to construct a single well-founded set that contains the arguments of both $f$ and $g$.

To show the termination of a system $f_1, f_2, f_3, \ldots, f_n$ of mutually recursive procedures, we resort to a more general method: We find (as before) a single well-founded set $W$ with an ordering $\prec$. In addition, we find a *termination function* $T_i$ corresponding to each procedure $f_i$, such that $T_i$ maps the arguments of $f_i$ into $W$ and such that, whenever a call $f_j(t)$ occurs in the execution of the procedure $f_i(x)$, we can establish the *termination condition*

$$T_j(t) < T_i(x) .$$

This suffices to prove the termination of the system, because if there were a computation containing an infinite sequence of calls

$$f_a(t_a) \text{ , } f_b(t_b) \text{ , } f_c(t_c) \text{ , } \ldots ,$$

the corresponding sequence

$$T_a(t_a) \text{ , } T_b(t_b) \text{ , } T_c(t_c) \text{ , } \ldots$$

of elements of $W$ would be infinitely decreasing, contradicting the definition of a well-founded set.

To illustrate this method, we will briefly consider this simple example of a system of mutually recursive procedures to compute the *gcd* of two nonnegative integers $x$ and $y$ :

```
$gcd_0(x\ y)$ <== if $x = 0$
                  then $y$
                  else $gcd_1(x\ y)$

$gcd_1(x\ y)$ <== if $y \geq x$
                  then $gcd_2(x\ y)$
                  else $gcd_3(x\ y)$

$gcd_2(x\ y)$ <== $gcd_1(x\ y-x)$

$gcd_3(x\ y)$ <== $gcd_0(y\ x)$ .
```

For this example, the naive approach is to show that the inputs $(x\ y)$ and the arguments of each procedure call belong to the well-founded set $W$ of pairs of nonnegative integers, and that the arguments of each procedure call are less than its inputs under some well-founded ordering, such as the lexicographic ordering. This approach fails here because, for instance, the main program $gcd_0(x\ y)$ executes a procedure call $gcd_1(x\ y)$ whose arguments are the same as the inputs.

It suffices, however, to take $W$ to be the set of triples of nonnegative integers, under the lexicographic ordering $<$. Corresponding to each procedure $gcd_i$ we have a termination function $T_i$ :

$T_0(x\ y) = (x\ y\ 2)$ ,

$T_1(x\ y) = (x\ y\ 1)$ ,

$T_2(x\ y) = (x\ y\ 0)$ , and

$T_3(x\ y) = (x\ y\ 0)$ .

Now, each time a procedure call $gcd_j(u\ v)$ is executed within a procedure $gcd_i(x\ y)$ we need to show the termination condition

$$T_j(u\ v) \prec T_i(x\ y) .$$

For example, because $gcd_0(x\ y)$ calls $gcd_1(x\ y)$ when $x$ is not zero, we have to show

$$(x\ y\ 1) \prec (x\ y\ 2) ,$$

which is clearly true under the lexicographic ordering. Because $gcd_3(x\ y)$ calls $gcd_0(y\ x)$ when $y$ is less than $x$ , we have to show

$$(y\ x\ 2) \prec (x\ y\ 0) ,$$

which also holds under the lexicographic ordering since $y < x$ .

# 4. STRUCTURE-CHANGING PROGRAMS

## A. Straight-Line Programs

The programs we have been developing up to now have been *structure-maintaining* programs: they do not alter the value of any variable or change the configuration of any data structure. Thus, any condition that is true before executing such a program will also be true afterwards. In this section, we extend the techniques we have already introduced to permit the construction of *structure-changing* programs; these programs can reset the values of variables, change the contents of an array, or alter the structure of a list or other data object. (Commonly, such changes are called *side effects*; this term has the unfortunate connotation that the effects are undesirable, rather like a headache.) In executing such a program, a condition that was previously false can be made true, and the opposite.

For example, a program that merely outputs the maximum element of an array is a structure-maintaining program; its execution does not change the contents of the array. On the other hand, a program to sort an array in place is a structure-changing program, because the contents of the array may be changed.

The basic principles of program construction introduced earlier (such as conditional formation, recursion formation, generalization, and procedure formation) extend naturally to the development of structure-changing programs. In addition, we will need some basic principles that specifically pertain to this new class of programs.

To express programming problems that require structure changing, we need to introduce new constructs into our specification language. To express programs that solve such problems, we need to introduce new primitive statements into our target language.

To the specification language we add the new construct

**achieve** $P$ ,

where $P$ is some condition. The meaning of this construct is that the corresponding program segment is to cause condition $P$ to become true. (Thus, **achieve** $x = 2$ can yield a program segment that sets $x$ to be 2.)

We also extend our target language to include assignment statements, such as *variable assignments*, e.g.,

$u \leftarrow t$ ,

*array assignments*, e.g.,

$a[i] \leftarrow t$,

and *list assignments*, e.g.,

$head(l) \leftarrow t$ and $tail(l) \leftarrow t$.

The effect of these statements is to change the value of the variable $u$, the contents of the array element $a[i]$, and the *head* and *tail* of the list $l$, respectively.

We will introduce other specification and target-language constructs in the context of specific examples.

Let us introduce rules that explicate the **achieve** construct and relate it to the assignment statements. For instance:

- The *achieve–elimination rule*

**achieve** $P$ => **prove** $P$ .

This rule expresses that to achieve some condition $P$, it suffices to prove that $P$ is already true. The rule is generally applied in conjunction with

- The *prove–elimination rule*

**prove** *true* => $\Lambda$ ,

where $\Lambda$ represents the empty program segment. Together, these rules allow us to remove from the program description any subexpression of form **achieve** $P$, where $P$ can be proven to be true. Because **prove** is a nonprimitive construct, a program segment containing a subexpression **prove** $P$ must be transformed until the subexpression is eliminated, i.e., until we prove that $P$ holds when control passes through the corresponding point.

- The *variable–assignment formation rule*

**achieve** $P(u)$ => **prove** $P(t)$
$u \leftarrow t$ for some $t$

where $u$ is a variable and $t$ is an expression. This rule expresses that if the condition $P(t)$ is true, we can achieve a condition of form $P(u)$ by the variable assignment $u \leftarrow t$.

Let us illustrate how these rules can be applied to construct a program to achieve $x = 2$. The specifications for the program are

$maketwo(x)$ <== **achieve** $x = 2$ .

Our top-level goal is therefore

**Goal 1:** **achieve** $x = 2$ .

Two of the above rules match this goal. The achieve-elimination rule transforms this goal into the subgoal

**Goal 2:** **prove** $x = 2$ ,

which fails. The variable-assignment formation rule, on the other hand, leads to the subgoal

**Goal 3:** **prove** $t = 2$
$x \leftarrow t$ for some $t$ .

Applying the rule for equality,

$u = u$ => *true* ,

forces us to take $t$ to be 2 itself; we obtain

**Goal 4:** **prove** *true*
$x \leftarrow 2$ .

Finally, the prove-elimination rule yields the ultimate program

*maketwo*($x$) <== $x \leftarrow 2$ .

## B. Conditional Programs

Let us illustrate how the conditional-formation rule extends to allow the introduction of tests into structure-changing programs. For this purpose, we will construct a program *sort*2($x$ $y$) to sort the values of two variables $x$ and $y$. We will assume that the target language contains the new instruction *interchange*($x$ $y$), which has the effect of exchanging the values of the variables $x$ and $y$. This instruction is described by the *interchange rule*

**achieve** $P(u\ v)$ => **prove** $P(v\ u)$
*interchange*($u$ $v$) ,

where $u$ and $v$ are variables.

The output specification for the *sort*2 program is *sort*2$(x\ y)$ <== **achieve** $x \le y$ *and perm*$((x_0\ y_0)(x\ y))$ .

Here, *perm*$((x_0\ y_0)(x\ y))$ means that the values of $x$ and $y$ are a permutation of their original values $x_0$ and $y_0$. [In the following, we will abbreviate this condition as *perm*$((x\ y))$.] This condition is necessary because, were it omitted, the *sort*2 program could achieve $x \le y$ simply by resetting $x$ and $y$, say to 1 and 2, respectively. However, the output specification for this program is to achieve two conditions at the same time; such goals require special treatment and will not be discussed until the next section. The purpose of this section is merely to illustrate conditional formation in structure-changing programs. Consequently, we will ignore the permutation property and pretend that the output specification has only the one condition, **achieve** $x \le y$. We will ensure that the permutation property is preserved by temporarily allowing *interchange*$(x\ y)$ to be the only structure-changing primitive in our target language.

Our top-level goal is therefore

**Goal 1:** **achieve** $x \le y$ .

The achieve-elimination rule,

**achieve** $P$ => **prove** $P$ ,

transforms this goal to form the subgoal

**Goal 2:** **prove** $x \le y$ .

We can neither prove nor disprove $x \le y$ -- $x$ and $y$ are inputs -- so we introduce a case analysis based on this condition.

**Case** $y < x$ : Here, we cannot achieve Goal 2, so we seek alternate ways to achieve Goal 1. Our interchange rule,

**achieve** $P(u\ v)$ => **prove** $P(v\ u)$
*interchange*$(u\ v)$ ,

causes us to transform Goal 1 into

**Goal 3:** **prove** $y \le x$
*interchange*$(x\ y)$ .

However, we are assuming that $y < x$ in this case. Therefore, the subexpression **prove** $y \le x$ is eliminated by applying the rule

$u \le v$ => *true* if $u < v$ ,

followed by the prove-elimination rule. Consequently, we generate the program segment

```
    interchange(x y)
```

in this case. It remains to consider the alternate case.

**Case** $x \leq y$ : Here, Goal 2, **prove** $x \leq y$, is achieved by the prove-elimination rule, and we are left with the empty program segment $\Lambda$ .

Our final program is therefore

```
    sort2(x y) <== if y < x
                   then interchange(x y)
                   else Λ
```

or, equivalently,

```
    sort2(x y) <== if y < x
                   then interchange(x y) .
```

## C. The Weakest-Precondition Operator

In formulating the specifications for the *sort*2 program in the previous section, we avoided including in the output specification the condition *perm*((*x y*)); otherwise, the top-level goal would have been

**achieve** $x \leq y$ *and perm*((*x y*)) .

Special difficulties arise in approaching a *simultaneous–goal problem*, i.e., a goal of the form

**achieve** $P_1$ *and* $P_2$ ,

where $P_1$ and $P_2$ are to hold simultaneously. We cannot always decompose such a goal into a sequence of two goals

**achieve** $P_1$
**achieve** $P_2$ ,

or

**achieve** $P_2$
**achieve** $P_1$ ,

because in the course of making the second condition true we may very well make the first false. For instance, in the *sort*2 problem, we can achieve $x \le y$ by setting $x$ to 1 and $y$ to 2, and we can achieve $perm((x\ y))$ by setting $x$ and $y$ to their original values, but no concatenation of these two programs will sort $x$ and $y$ .

To handle such simultaneous-goal problems properly, we need to analyze what effect a given program segment has on the truth of a given condition. For this purpose, we define the concept of the *weakest precondition*; we will then use this concept to formulate a program-modification technique that will serve as the basis for our *simultaneous-goal principle*.

If $S$ is a program segment and $P$ is a condition, we define the *weakest precondition* $wp(S\ P)$ to be the condition $P'$ such that

$P'$ is true before executing $S$<br>
if and only if<br>
$P$ is true afterwards.

(We will assume throughout that $S$ terminates.) We will also call $wp(S\ P)$ the result of *passing P back over S*. Thus, the weakest precondition for the execution of the program segment $x \leftarrow x+1$ to achieve the condition $x \ge 2$ is $x+1 \ge 2$, i.e., $x \ge 1$. In other words,

$$wp(\ x \leftarrow x+1 \quad x \ge 2\ ) \text{ is } x \ge 1\ .$$

We can represent the properties of the weakest-precondition operator by transformation rules. Some of these rules tell how to compute the weakest precondition for particular specification- or target-language constructs:

$$wp(\ \Lambda \quad P\ ) \Rightarrow P$$

$$wp(\ u \leftarrow t \quad P(u)\ ) \Rightarrow P(t)$$

$$wp(\ interchange(u\ v) \quad P(u\ v)\ ) \Rightarrow P(v\ u)$$

$$wp(\ \textit{if } q \textit{ then } S_1 \textit{ else } S_2 \quad P\ ) \Rightarrow (\textit{if } q \textit{ then } wp(S_1\ P)) \textit{ and } (\textit{if not } q \textit{ then } wp(S_2\ P))$$

$$wp(\ \textit{if } q \textit{ then } S \quad P\ ) \Rightarrow (\textit{if } q \textit{ then } wp(S\ P)) \textit{ and } (\textit{if not } q \textit{ then } P)$$

$$wp(\ S_1 S_2 \quad P\ ) \Rightarrow wp(\ S_1 \quad wp(S_2\ P)\ )$$

$$wp(\ \textbf{achieve } Q \quad P\ ) \Rightarrow true \qquad \text{if } Q \text{ implies } P\ .$$

The weakest-precondition rule for the recursion construct does not tell us how to compute the weakest precondition, but only how to prove by mathematical induction that a given condition is indeed the weakest precondition for a recursive call. Suppose that $f(s)$ is a call to a procedure

$$f(x) \Longleftarrow B(x) \,,$$

and that $\prec$ is a well-founded ordering. Then, for any condition $P(x)$, we have

$$wp(f(s)\ \ P(s)) = P'(s)$$

if we can prove

$$wp(B(x)\ \ P(x)) = P'(x)$$

under the inductive assumption that

$$wp(f(t)\ \ P(t)) = P'(t)$$

for any $t$ such that $t \prec x$. (Often, $\prec$ is taken to be the well-founded ordering used to prove the termination of $f$.)

In addition to rules that give the weakest preconditions for the various programming-language constructs, there are rules for computing the weakest preconditions for specific conditions. For example,

$$wp(S\ \ true) \Rightarrow true \,,$$

$$wp(S\ \ false) \Rightarrow false \,,$$

$$wp(S\ \ P_1\ and\ P_2) \Rightarrow wp(S\ P_1)\ and\ wp(S\ P_2) \,,$$

$$wp(S\ \ P_1\ or\ P_2) \Rightarrow wp(S\ P_1)\ or\ wp(S\ P_2) \,, \text{ and}$$

$$wp(S\ \ not\ P) \Rightarrow not\ wp(S\ P) \,.$$

When a new construct is defined in terms of other constructs, we can often deduce the weakest-precondition rule for the new construct. For example, *sort*2($u$ $v$) is the program

```
if v < u
then interchange(u v) .
```

Therefore,

$$\begin{aligned}
&wp\ (\ sort2(u\ v)\ \ P(u\ v)\ ) \\
&= wp\ (\ if\ v < u\ then\ interchange(u\ v)\ \ P(u\ v)\ ) \\
&= if\ v < u\ then\ wp(interchange(u\ v)\ \ P(u\ v))\ and \\
&\quad if\ u \leq v\ then\ P(u\ v) \\
&= if\ v < u\ then\ P(v\ u)\ and \\
&\quad if\ u \leq v\ then\ P(u\ v)\ .
\end{aligned}$$

We thus obtain the *sort*2 rule

$$\begin{aligned}
wp(\ sort2(u\ v)\ \ P(u\ v)\ ) => &(if\ v < u\ then\ P(v\ u))\ and \\
&(if\ u \leq v\ then\ P(u\ v))\ .
\end{aligned}$$

On the other hand, if we introduce a new construct into our specification or target language that is not expressed in terms of other constructs, we must also provide weakest-precondition rules for the new construct. For example, we have used the construct *perm*($l$) to denote that the values of the variables in a list $l$ are a permutation of their original values; we must therefore introduce rules such as

$$wp(\ interchange(u\ v)\ \ perm(l)\ ) => perm(l) \quad \text{if } u \text{ and } v \text{ belong to } l\ .$$

In other words, interchanging the values of two of the variables of the list does not affect the permutation property. Similarly, we will introduce the construct *only l changed* to denote that no variables other than those in $l$ are changed by the program segment; we will also introduce the corresponding rule

$$wp(\ u \leftarrow t \quad only\ l\ changed\ ) => only\ l\ changed \quad \text{if } u \in l\ .$$

The weakest-precondition operator is used to express many transformation rules that manipulate structure-changing programs. Two *regression rules* are obtained directly from the definition of the weakest precondition:

$$\begin{array}{lll}
S & => & \textbf{prove}\ wp(S\ P) \\
\textbf{prove}\ P & & S
\end{array}$$

and

$$\begin{array}{lll}
S & => & \textbf{achieve}\ wp(S\ P) \\
\textbf{achieve}\ P & & S\ .
\end{array}$$

That is, to prove or achieve a condition $P$ after a program segment $S$, one may just as well prove or achieve the weakest precondition $wp(S\ P)$ before $S$.

We have two additional rules for pushing goals back into conditional expressions:

```
(if $q$          =>   if $q$
 then $S_1$            then $S_1$
 else $S_2$)                achieve $P$
achieve $P$           else $S_2$
                           achieve $P$
```

and (consequently)

```
(if $q$          =>   if $q$
 then $S_1$)           then $S_1$
achieve $P$                 achieve $P$
                      else achieve $P$ .
```

Let us see how these concepts can be applied to obtain a systematic program-modification technique, which will eventually be used in the simultaneous-goal rule.

The weakest-precondition operator of Dijkstra [1975] was motivated by the program-verification technique of Floyd [1967] and Hoare [1969].

## D. A Program-Modification Technique

Imagine that we have a program segment $S$ that is a concatenation $S_1S_2$ of two instructions. Suppose we wish to alter $S$ to achieve some new condition $P$. The most straightforward approach is to add new instructions to the end of $S$ that achieve the new condition; we may describe the desired modification as

```
$S_1$
$S_2$
achieve $P$ .
```

However, according to the regression rule of the previous section, we may just as well add new instructions to achieve $wp(S_2\ P)$ before $S_2$; i.e., we can pass $P$ back over $S_2$, yielding

```
$S_1$
achieve $wp(S_2\ P)$
$S_2$ .
```

Similarly, we can pass $wp(S_2\ P)$ back over $S_1$ :

```
achieve $wp(S_1\ wp(S_2\ P))$
$S_1$
$S_2$ .
```

Thus, we can make modifications at any point in $S$ to achieve the desired condition.

For example, suppose that $S$ is a program segment

```
$y \leftarrow x$
$y \leftarrow y+1$
```

and that we want to modify $S$ to achieve the relation $y \geq 2$; this modification task may be expressed as

```
$y \leftarrow x$
$y \leftarrow y+1$
achieve $y \geq 2$ .
```

We can certainly achieve the new condition by adding an instruction (e.g., $y \leftarrow 2$) to the end of the program. But, by the regression rule, we can also transform the above task into

```
$y \leftarrow x$
achieve $y \geq 1$
$y \leftarrow y+1$
```

and then into

```
achieve $x \geq 1$
$y \leftarrow x$
$y \leftarrow y+1$ .
```

[In the first transformation, we relied on the fact that $wp(\ y \leftarrow y+1\ \ y \geq 2\ )$ is $y+1 \geq 2$ , i.e., $y \geq 1$ ; the second step relied on the fact that $wp(\ y \leftarrow x\ \ y \geq 1\ )$ is $x \geq 1$.] Thus, we can also perform the required modification by adding instructions in the middle of the program (e.g., $y \leftarrow 1$) or at the beginning (e.g., $x \leftarrow 1$) .

Of course, a program segment modified by the above technique may no longer achieve the purpose for which it was originally intended. Suppose that a program segment $S$ was originally intended to achieve some condition $P_1$, and we want to modify $S$ to achieve a new condition $P_2$ as well as the original condition $P_1$. To ensure that the modified program still achieves its original purpose, we *protect* $P_1$ at the end of $S$ during the modification process. This modification task is denoted by

```
$S$
achieve $P_2$
protect $P_1$ .
```

The purpose of the *protection condition* **protect** $P_1$ is to block any modification that does not allow us subsequently to prove the protected condition $P_1$. Let us see how such a protection condition is checked.

Returning to the previous example, suppose in modifying the program segment

```
$y \leftarrow x$
$y \leftarrow y+1$
```

to achieve the new condition $y \geq 2$, we want to protect the condition $x < y$ that the program originally achieved. Our task can thus be described as

```
Goal 1:  $y \leftarrow x$
         $y \leftarrow y+1$
         achieve $y \geq 2$
         protect $x < y$ .
```

We have seen that we can achieve the desired condition $y \geq 2$ by introducing statements at the end (e.g., $y \leftarrow 2$), the middle (e.g., $y \leftarrow 1$), or the beginning (e.g., $x \leftarrow 1$) of the program. To check the protection condition for a proposed modification, we try to prove that the protected condition still holds in the modified program. Thus, to see whether introducing $y \leftarrow 2$ at the end of the program violates the protected condition, we establish the subgoal

```
Goal 2:  $y \leftarrow x$
         $y \leftarrow y+1$
         $y \leftarrow 2$
         prove $x < y$ .
```

This means that we must prove that $x < y$ holds after the execution of the modified program.

In fact, we fail to prove this condition, so the proposed modification is rejected. Similarly, we cannot achieve the desired condition by inserting the statement $y \leftarrow 1$ in the middle of the program, because we fail to establish the corresponding subgoal

```
Goal 3:  $y \leftarrow x$
         $y \leftarrow 1$
         $y \leftarrow y+1$
         prove $x < y$ .
```

However, the third proposed modification, to insert $x \leftarrow 1$ at the beginning of the program, does maintain the protected condition:

```
Goal 4:  $x \leftarrow 1$
         $y \leftarrow x$
         $y \leftarrow y+1$
         prove $x < y$ .
```

Let us see in more detail how such a proof is conducted.

Applying the regression rule

```
$S$          =>    prove $wp(S\ P)$
prove $P$          $S$ ,
```

we develop the subgoal

```
Goal 5:  $x \leftarrow 1$
         $y \leftarrow x$
         prove $wp(\ y \leftarrow y+1\ \ x < y\ )$
         $y \leftarrow y+1$ .
```

The weakest-precondition rule for assignment statements,

$$wp(\ u \leftarrow t\ \ P(u)\ ) \Rightarrow P(t)\ ,$$

eliminates the weakest-precondition operator:

```
Goal 6:  $x \leftarrow 1$
         $y \leftarrow x$
         prove $x < y+1$
         $y \leftarrow y+1$ .
```

Again applying the regression and assignment rules, we obtain

```
Goal 7:  $x \leftarrow 1$
         prove $x < x+1$
         $y \leftarrow x$
         $y \leftarrow y+1$ .
```

The condition **prove** $x < x+1$ can now be established by the rule

$$u < u+1 \Rightarrow \mathit{true} \quad \text{if } u \text{ is a number.}$$

Having verified the protection condition, we obtain the program

```
x ← 1
y ← x
y ← y + 1 ,
```

which achieves both the original condition $x < y$ and the additional condition $y \geq 2$.

The previous discussion neglected the strategic aspects of our program modification technique. How do we divide our time between altering the program to achieve a new condition $P_2$ and ensuring that a protected condition $P_1$ is still achieved? The most adventurous strategy is first to complete the modification necessary to achieve $P_2$, and then to check that $P_1$ still holds. This can be wasteful, however, because we may need to do a lot of work modifying the program to achieve $P_2$ before we discover that $P_1$ is not achieved by the modified program. A more conservative strategy is to check that the protection conditions are maintained each time a new instruction is inserted during the modification process; thus a proposed modification that does not achieve $P_1$ may be rejected quite early. For example, if $P_1$ is the permutation property $perm(l)$, that the values of the variables in the list $l$ are to be a permutation of their original values, we will admit modifications that interchange the values of variables in $l$, but reject modifications that attempt to assign new values to these variables. This conservative strategy is adhered to by our implemented system; it is a bit too restrictive, because a modification that satisfies the protection condition only at the final stage may be rejected if its protection condition is checked prematurely.

The above modification technique allows us to insert new instructions into the program segment, but not to alter or delete any of the instructions that are already there. Such modifications may sometimes be necessary, but they are beyond the scope of our technique.

The protection concept was used by Sussman [1975] as an approach to plan formation by the successive debugging of nearly correct plans.

## E. The Simultaneous-Goal Principle

We have remarked that when faced with a simultaneous-goal problem

achieve $P_1$ *and* $P_2$ ,

we cannot decompose the goal into the linear sequence

achieve $P_1$
achieve $P_2$

because, in the course of making $P_2$ true, we may be making $P_1$ false. For the same reason, it is not enough to reverse the order in which the goals are achieved. However, the program modification technique of the previous section gives us a way of solving such a problem. To apply this technique, we first construct a program that achieves $P_1$; we then modify this program to achieve $P_2$ while protecting $P_1$. The *simultaneous-goal rule* that represents this approach is

```
achieve P_1 and P_2 =>  achieve P_1
                        achieve P_2
                        protect P_1 .
```

(Of course, the roles of $P_1$ and $P_2$ can be reversed.) This rule extends naturally to the more general problem of achieving many conditions simultaneously; we consider $P_1$ to be one of the conditions, and $P_2$ to be the conjunction of all the others.

The simultaneous-goal principle does not dictate which condition we attempt to achieve first. In general, if we discover that one of the conditions is already true, we prefer to "achieve" that condition first, protect it, and go on to achieve the others. Furthermore, we may have rules for specific subject domains that cause these conditions to be reordered.

Let us see how the simultaneous-goal rule applies to a new sorting problem; this time we wish to sort three variables $x$, $y$, and $z$. The problem can be specified by

*sort3*$(x\ y\ z)$ <== **achieve** $x \le y$ *and* $y \le z$ *and perm*$((x\ y\ z))$
**where** $x$, $y$, and $z$ are variables with numerical values.

We will introduce the program *sort2*$(u\ v)$, which we constructed in the previous section, as a primitive in the target language. Because the *sort2* program was constructed to achieve the condition $u \le v$, we can include the *sort2-formation* rule

**achieve** $u \le v$ => *sort2*$(u\ v)$

in our set of transformation rules. Because *sort2*$(u\ v)$ was specified to maintain the condition *perm*$((u\ v))$, we can add the *sort2-perm rule*

*wp*( *sort2*$(u\ v)$ *perm*$(l)$ ) => *perm*$(l)$ if $u$ and $v$ belong to $l$.

The top-level goal for the *sort3* derivation is

**Goal 1:** **achieve** $x \le y$ *and* $y \le z$ *and perm*$((x\ y\ z))$ .

We apply the simultaneous-goal principle; because the condition *perm*$((x\ y\ z))$ is already true, it is the first to be "achieved":

**Goal 2:** **achieve** *perm*(⟨*x y z*⟩)
**achieve** *x* ≤ *y and y* ≤ *z*
**protect** *perm*(⟨*x y z*⟩) .

Because *perm*(⟨*x y z*⟩) is true initially, we can eliminate the first task **achieve** *perm*(⟨*x y z*⟩) by applying first the achieve-elimination rule

**achieve** *P* => **prove** *P* ,

and later the prove-elimination rule

**prove** *true* => Λ .

We obtain

**Goal 3:** **achieve** *x* ≤ *y and y* ≤ *z*
**protect** *perm*(⟨*x y z*⟩) .

The first task, **achieve** *x* ≤ *y* and *y* ≤ *z*, is another simultaneous-goal problem; we again apply the simultaneous-goal rule, arbitrarily attempting to achieve the condition *x* ≤ *y* first.

**Goal 4:** **achieve** *x* ≤ *y*
**achieve** *y* ≤ *z*
**protect** *x* ≤ *y*
**protect** *perm*(⟨*x y z*⟩) .

Applying the new *sort*2-formation rule

**achieve** *u* ≤ *v* => *sort*2(*u v*)

to the first task, **achieve** *x* ≤ *y*, yields

**Goal 5:** *sort*2(*x y*)
**achieve** *y* ≤ *z*
**protect** *x* ≤ *y*
**protect** *perm*(⟨*x y z*⟩) .

We first attempt to apply the same rule to the second task, **achieve** *y* ≤ *z*, yielding

**Goal 6:** *sort*2(*x y*)
*sort*2(*y z*)
**protect** *x* ≤ *y*
**protect** *perm*(⟨*x y z*⟩) .

However, in executing the instruction *sort*2(*y z*) we may violate the protected condition $x \le y$ . (In particular, if *z* was initially the smallest of the three values, then sorting *y* and *z* makes *y* the smallest: *x* and *y* will now be out of order.) Therefore, we are forced to backtrack and consider alternate means for achieving Goal 5.

By applying the regression rule

```
S               =>  achieve $wp(S\ P)$
achieve $P$          $S$ ,
```

we derive

```
Goal 7:   achieve $wp(\ sort2(x\ y)\ \ y \le z\ )$
          $sort2(x\ y)$
          protect $x \le y$
          protect $perm((x\ y\ z))$ .
```

We have already derived the weakest-precondition rule for the *sort*2 instruction; it is

$$wp(\ sort2(u\ v)\ \ P(u\ v)\ ) \Rightarrow (\textit{if } v < u \textit{ then } P(v\ u)) \textit{ and}$$
$$(\textit{if } u \le v \textit{ then } P(u\ v))\ .$$

Applying this rule produces

```
Goal 8:   achieve $(if\ y < x\ then\ x \le z)\ and$
                  $(if\ x \le y\ then\ y \le z)$
          $sort2(x\ y)$
          protect $x \le y$
          protect $perm((x\ y\ z))$ .
```

Intuitively, the first task of this goal,

```
achieve   $(if\ y < x\ then\ x \le z)\ and$
          $(if\ x \le y\ then\ y \le z)$ ,
```

is to achieve that the value of *z* is the largest of the three values: if this condition holds before *sort*2(*x y*) is executed, we know that the desired condition $y \le z$ will be true afterwards. This task is still another simultaneous-goal problem, and is achieved by another application of the simultaneous-goal principle. We will not describe in detail how this task is accomplished. The resulting program segment is

```
$if\ y < x\ then\ sort2(x\ z)$
$if\ x \le y\ then\ sort2(y\ z)$ .
```

The corresponding goal is

```
Goal 9:   if $y < x$ then sort2($x$ $z$)
          if $x \leq y$ then sort2($y$ $z$)
          sort2($x$ $y$)
          protect $x \leq y$
          protect perm(($x$ $y$ $z$)) .
```

It remains to check the protection conditions. Intuitively, the first condition $x \leq y$ is satisfied because it occurs immediately after the *sort*2($x$ $y$) instruction, which achieves this relation. The second condition *perm*(($x$ $y$ $z$)) holds because it is true initially and it is preserved by the three *sort*2 instructions in the program. In practice, these conditions would be established by application of the regression and weakest-precondition rules. (As we remarked, our implementation checks these conditions repeatedly while the program is being modified rather than waiting until the end of the derivation.)

The final program we obtain is

```
sort3($x$ $y$ $z$) <== if $y < x$ then sort2($x$ $z$)
                       if $x \leq y$ then sort2($y$ $z$)
                       sort2($x$ $y$) .
```

This concludes our discussion of the simultaneous-goal rule; we will see further applications of this rule in the next section, in the synthesis of a somewhat less trivial program.

> An extended discussion of the simultaneous-goal problem appears in Waldinger [1977]. A similar approach to the problem was devised by Warren [1974], but he did not use the weakest-precondition operator. Other methods have been applied to the problem by Sacerdoti [1975] and Tate [1975].

## F. Recursive Programs

The structure-changing programs we have constructed so far contain no recursive calls. Our next example illustrates how the recursion-formation techniques we have introduced earlier can be applied to structure-changing programs.

We are asked to construct a program to find the maximum *max*($a$ $n$) of an array segment $a[0 : n]$, the list of $n+1$ elements $a[0]$, $a[1]$, ..., $a[n]$. The specifications for this program may be written as

$max(a\ n)$ <== **achieve** $all(a[0:n]) \le z$ *and*
$z \in a[0:n]$ *and*
*only* $z$ *changed*

**where** $a$ is an array of numbers and
$n$ is an integer and
$0 \le n$ .

Recall that *only* $z$ *changed* means that no variable other than $z$ can be changed by the program; in particular, this condition ensures that the final program will have no surprising side effects, and that it will not satisfy its specifications perversely, say by setting $z$ and all the elements of the array segment to zero.

Our top-level goal is thus

**Goal 1:** **achieve** $all(a[0:n]) \le z$ *and*
$z \in a[0:n]$ *and*
*only* $z$ *changed* .

This goal has the form of a simultaneous-goal problem. The third condition, *only* $z$ *changed*, is of course true initially, so we decide to "achieve" it first; it will then be eliminated by the achieve- and prove-elimination rules. The other two conditions may be approached in either order. We obtain

**Goal 2:** **achieve** $all(a[0:n]) \le z$
**achieve** $z \in a[0:n]$
**protect** $all(a[0:n]) \le z$
**protect** *only* $z$ *changed* .

Assume that we have the following three transformation rules that relate the *all* construct and the array segment:

- The *vacuous rule*

  $P(all(a[u:w])) \Rightarrow true$ if $u > w$

  (any condition is true for every element of the empty segment),

- The *singleton rule*

  $P(all(a[u:w])) \Rightarrow P(a[u])$ if $u = w$

  (a condition is true of every element of a "singleton" segment if the condition holds for that segment's sole element), and

- The *decomposition rule*

$$P(all(a[u : w])) \Rightarrow P(all(a[u : w-1])) \text{ and } P(a[w]) \quad \text{if } u < w$$

(a condition is true for every element of a segment containing two or more elements if the condition holds for the final element of the segment as well as for every element of the initial segment).

We focus our attention on the first task in Goal 2:

**Goal 3: achieve** $all(a[0 : n]) \leq z$ .

The three *all* rules each match this goal. The vacuous rule requires that the segment be empty; we know this is false by the condition $0 \leq n$ in the input specification. The singleton rule requires that the segment have but one element, i.e., that $0 = n$; we cannot prove or disprove this condition, so we make it the basis for a case analysis.

**Case** $0 \neq n$ (i.e., $0 < n$) : Here, the singleton rule fails, but the decomposition rule, which actually requires that $0 < n$, succeeds in decomposing the goal into the conjunction of two conditions. These conditions may be treated separately by the simultaneous-goal principle, yielding

**Goal 4: achieve** $all(a[0 : n-1]) \leq z$
**achieve** $a[n] \leq z$
**protect** $all(a[0 : n-1]) \leq z$ .

We will consider the three tasks of this goal in turn. The first task, to achieve

$$all(a[0 : n-1]) \leq z \text{ ,}$$

is an instance of one of the conditions of the top-level goal; therefore, the recursion-formation rule proposes achieving it by means of a recursive call $max(a\ n-1)$. The input and termination conditions for this call are straightforward.

We now focus our attention on the second task of Goal 4,

**Goal 5: achieve** $a[n] \leq z$ .

Before attempting to achieve a condition, the achieve-elimination rule always tries to determine whether that condition is already true; we can neither prove nor disprove it, so we make it the basis for a further case analysis.

**Case** $z < a[n]$ : In this case, we must seek alternate means to achieve Goal 5. Recall that we have a variable-assignment formation rule

**achieve** $P(u)$ => **prove** $P(t)$
$u \leftarrow t$ for some $t$

where $u$ is a variable and $t$ is an expression. Taking $P(u)$ to be $a[n] \leq u$, $t$ to be $a[n]$, and $u$ to be $z$, we can achieve Goal 5 by the assignment statement

$z \leftarrow a[n]$ ,

because $a[n] \leq a[n]$.

[Note that we could also achieve Goal 5 by the array-assignment rule

$a[n] \leftarrow z$ ,

or the *sort*2 instruction

*sort*2($a[n]$ $z$) ;

these solutions would be rejected, however, because they violate the protected condition *only z changed*.]

**Case** $a[n] \leq z$ : Here, the condition of Goal 5 is already true, and can be "achieved" by the empty program.

We have achieved Goal 5 in both cases; the conditional-formation principle yields the program

*if* $z < a[n]$ *then* $z \leftarrow a[n]$ .

We have thus completed the second task of Goal 4.

We now proceed to consider the third task, which is to check the protection condition

**Goal 6:** *max*($a$  $n-1$)
*if* $z < a[n]$
*then* $z \leftarrow a[n]$
**prove** *all*($a[0:n-1] \leq z$) .

Applying the prove-regression rule

$S$ => **prove** $wp(S\ P)$
**prove** $P$ $S$ ,

the weakest-precondition rule for the *if–then* construct

$$wp(\ \textit{if } q \textit{ then } S \quad P\ ) \Rightarrow \textit{if } q \textit{ then } wp(S\ P) \textit{ and}$$
$$\textit{if not } q \textit{ then } P\ ,$$

and the weakest-precondition rule for the assignment statement

$$wp(\ u \leftarrow t \quad P(u)\ ) \Rightarrow P(t)\ ,$$

we obtain

```
Goal 7:  $max(a\ n-1)$
         prove $if\ z < a[n]\ then\ all(a[0:n-1]) \le a[n]\ and$
                 $if\ a[n] \le z\ then\ all(a[0:n-1]) \le z$
         $if\ z < a[n]$
         $then\ z \leftarrow a[n]$ .
```

Note that *max*($a\ n$) was specified to achieve the condition

$$all(a[0:n]) \le z\ ;$$

therefore, by mathematical induction, the recursive call $max(a\ n-1)$ can be assumed to achieve

$$all(a[0:n-1]) \le z\ .$$

The second condition we are asked to prove,

$$\textit{if } a[n] \le z \textit{ then } all(a[0:n-1]) \le z\ ,$$

follows at once. The first condition,

$$\textit{if } z < a[n] \textit{ then } all(a[0:n-1]) \le a[n]\ ,$$

follows directly by the transitive rule.

This completes the final task of Goal 4, and thus we have achieved the condition of Goal 3, that $all(a[0:n]) \le z$, for the case where $0 < n$. The remaining case is more easily disposed of.

**Case** $n = 0$ : Here, the segment $a[0:n]$ has only one element, and the singleton rule reduces Goal 3 to the following:

**Goal 8: achieve** $a[0] \le z$ .

This condition is achieved by the assignment statement

$$z \leftarrow a[0]\ ,$$

as before.

We have constructed program segments that achieve Goal 3 in each case; the resulting conditional segment is

```
if $n = 0$
then $z \leftarrow a[0]$
else $max(a\ n-1)$
     if $z < a[n]$
     then $z \leftarrow a[n]$ .
```

There are three additional tasks in Goal 2 that we must perform: We must achieve the condition

$$z \in a[0:n]\ ;$$

this condition is already true, and may be proved by application of the regression and weakest-precondition rules. Next, we must check that the protected condition

$$all(a[0:n]) \leq z$$

is satisfied; this is true, because we have just constructed a segment that achieves this condition, and in "achieving" the additional condition $z \in a[0:n]$ we made no changes to this segment. Finally, we must ensure that the protected condition

*only z changed*

is satisfied; this is true, because only assignments to $z$ occur in the program we have constructed.

Having established the protection conditions, we are left with the final program

```
$max(a\ n)$ <== if $n = 0$
                then $z \leftarrow a[0]$
                else $max(a\ n-1)$
                     if $z < a(n)$
                     then $z \leftarrow a[n]$ .
```

## G. The Modification of Recursive Programs

The program-modification technique we introduced for loop-free programs extends naturally to permit the modification of recursive structure-changing programs.

Assume we are given the program *max(a n)* constructed in the preceding section; this program finds the value of the maximum element in an array. Suppose that we wish to extend that program to obtain a new program *maxindex(a n)* for finding the index of that maximum element as well as its value. In other words, we want to modify the program *max* to achieve the new condition

$$a[y] = z \textit{ and } 0 \le y \le n$$

while protecting the original condition

$$all(a[0 : n]) \le z \textit{ and } z \in a[0 : n]$$

that the program was intended to achieve. Note that we do not protect the condition *only z changed* that the program originally achieved; this is because we want to change the value of $y$ as well as $z$. Instead, we include

$$\textit{only } y, z \textit{ changed}$$

among the new conditions to be achieved by *maxindex*.

Our modification task is thus specified as follows:

```
maxindex(a n) <== if $n = 0$
                  then $z \leftarrow a[0]$
                  else maxindex($a$ $n-1$)
                       if $z < a[n]$
                       then $z \leftarrow a[n]$
                  achieve $a[y] = z$ and $0 \le y \le n$ and only $y, z$ changed
                  protect $all(a[0 : n]) \le z$ and $z \in a[0 : n]$

where $a$ is an array of numbers and
      $n$ is an integer and
      $0 \le n$ .
```

Here, we have replaced the recursive calls to *max*, the old program, by recursive calls to the extended program *maxindex*. Goal 1 is formed directly from these specifications, and will not be copied here.

Note that it is quite necessary to protect the condition $all(a[0 : n]) \le z$; otherwise, we could achieve the new conditions by perversely resetting $z$ to $a[0]$ and setting $y$ to 0. The second condition, on the other hand, is actually redundant; if $a[y] = z$ and $0 \le y \le n$, then certainly $z \in a[0 : n]$. Applying the usual regression and weakest-precondition rules, we derive

```
Goal 2:  if $n = 0$
         then achieve $a[y] = a[0]$ and $0 \le y \le n$ and only $y, z$ changed
             $z \leftarrow a[0]$
         else maxindex($a$ $n-1$)
             if $z < a[n]$
             then achieve $a[y] = a[n]$ and $0 \le y \le n$ and only $y, z$ changed
                 $z \leftarrow a[n]$
             else achieve $a[y] = z$ and $0 \le y \le n$ and only $y, z$ changed
         protect $all(a[0:n]) \le z$ and $z \in a[0:n]$ .
```

The task

**achieve** $a[y] = a[0]$ *and* $0 \le y \le n$ *and only* $y, z$ *changed* ,

which occurs in the branch for which $n = 0$, is found to be achieved by the assignment

$y \leftarrow 0$ ,

by application of the simultaneous-goal principle and the variable-assignment formation rule. Similarly, the task

**achieve** $a[y] = a[n]$ *and* $0 \le y \le n$ *and only* $y, z$ *changed* ,

which occurs after the recursive call in the case $z < a[n]$, is found to be achieved by the assignment

$y \leftarrow n$ .

Finally, the task

**achieve** $a[y] = z$ *and* $0 \le y \le n$ *and only* $y, z$ *changed* ,

occurs immediately after the recursive call *maxindex*($a$ $n-1$) in the case $a[n] \le z$. The recursive call can be assumed inductively to achieve the condition

$a[y] = z$ *and* $0 \le y \le n-1$ *and only* $y, z$ *changed* ;

thus, the desired condition is already true.

The protected condition

$all(a[0:n]) \le z$ *and* $z \in a[0:n]$ ,

which was achieved by our original program *max*($a$ $n$), has not been affected by any of our modifications; the only instructions we have added are assignments to $y$. The final *maxindex* program we obtain is thus

```
maxindex(a n) <== if n = 0
                  then y ← 0
                       z ← a[0]
                  else maxindex(a n−1)
                       if z < a[n]
                       then y ← n
                            z ← a[n] .
```

The modification of recursive programs can be initiated by the simultaneous-goal principle if the program constructed to achieve one of the goal conditions happens to be recursive. However, modification of a given program may also be regarded as an independent programming task; this application is discussed further in Section 5C.

# 5. IMPLICATIONS FOR PROGRAMMING METHODOLOGY

In program synthesis as we have defined it, a person formulates the purpose of the program he wants without indicating a procedure to achieve that purpose. In practice, even the most computationally naive user of a program-synthesis system is likely to have some idea of an algorithm that could be employed by the desired program. This algorithm may not be entirely satisfactory: it may not achieve all the desired conditions, it may be incompletely specified, or it may lead to an inefficient program. Nevertheless, it would be foolish to prevent the user from conveying this information to the system, because it is easier to derive a program from a partially specified algorithm than from a specification that expresses only the program's purpose. In this section, we will show how the program-synthesis techniques we have already introduced can be applied to transform a partially specified procedure into a complete program.

Actually, we have already seen some examples in which the specifications had a procedural component. In the *maxindex* example (Section 4G), our specifications were given in the form of a complete *max* program with some additional conditions to be achieved. In the *reverse* example (Section 3C), the specifications were composed of a complete *reverse* program, which was transformed into a more efficient equivalent. These examples were introduced to illustrate particular program-synthesis techniques. The emphasis in this section will be on the actual task performed.

We will consider separately three ways in which the procedural components of a specification can be presented.

- *Program transformation.* The specifications are given in the form of a clear--perhaps inefficient--program, which is then transformed into an efficient--perhaps unclear--equivalent.
- *Data abstraction.* The specifications are given in the form of a complete program that operates on certain *abstract data types*, structures (such as sets, stacks, or graphs) whose properties are expressed precisely but whose machine representation is unspecified; the program is then transformed to replace each operation on the abstract data types by a corresponding concrete operation on a chosen machine representation.
- *Program modification.* We are given a complete program that performs one task successfully; we wish to extend the program to achieve an additional condition, while still performing its original task.

Although we consider each of these topics separately, the same techniques can be applied to transform a procedure whose description is subject to all three modes of imprecision. In other words, the given specifications could present an inefficient procedure, expressed in terms of abstract structures, that needs to be extended to achieve additional conditions. Of course, there are other ways in which the description may be imprecise besides the three we will discuss here.

## A. Transformation: Programs => Better Programs

Often the clearest, simplest program for a given task may not be the most efficient; if we attempt to construct an efficient program for the task at once, our result is likely to be unclear, and perhaps incorrect as well. It has been suggested, therefore, that we construct our program in two stages: we begin by setting efficiency considerations aside for awhile; we construct as clear and straightforward a program as possible. We then transform this program to make it more efficient, possibly losing some clarity during the process.

It is argued that the programs produced in this way are more likely to be correct than programs produced by the conventional one-phase method. The first version is likely to be correct by virtue of its clarity; the second version is produced by the application of transformation rules that preserve the correctness of the first version while improving its efficiency.

We have already seen program-synthesis techniques applied to a transformation problem, in Section 3C. In that example, we were given the following program for reversing a list:

```
reverse(l) <== if empty(l)
               then nil
               else append(reverse(tail(l))
                                list(head(l)) ) .

append($l_1$ $l_2$) <== if empty($l_1$)
                        then $l_2$
                        else cons(head($l_1$)
                                  append(tail($l_1$) $l_2$)) .
```

Treating this program itself as the specifications, we developed the following system of two programs for performing the same task:

```
reverse(l) <==  if empty(l)
                then nil
                else reversegen(l nil) ,
```

where

```
reversegen(l m) <== if empty(tail(l))
                    then cons(head(l) m)
                    else reversegen(tail(l)
                                    cons(head(l) m)) .
```

The original *reverse* program is quite inefficient: each execution may require many calls to the *append* program; each of these calls to *append* produces a new copy of its first argument. On the other hand, in the final system of programs, the expensive *append* operation is replaced by the economical *cons*. Furthermore, the recursion is of a special form that can be evaluated without the use of a stack; in fact, this system can be converted to the following iterative *reverse* program by application of a recursion-removal transformation rule

```
reverse(l) <== if empty(l)
               then output(nil)
               else m ← nil
                    while not empty(tail(l))
                    do  m ← cons(head(l) m)
                        l ← tail(l)
                    output(cons(head(l) m)) .
```

By exploiting the properties of the operations in the original *reverse* program, we have managed to transform it to a more efficient program that achieves the same purpose by a fundamentally different method.

In this example, our specifications were given in the form of a complete program, with no other indication of the purpose to be achieved. We were fortunate to perform the same task by an entirely different and more efficient method. In general, if the specification of the program is purely procedural, such radical improvements are difficult to achieve; in omitting any statement of purpose from the given specification, we are biased toward adopting the algorithm of the given program, instead of seeking to achieve the same purpose in a new way.

For example, suppose that we want to construct a program to sort a list of numbers. Our description of the desired program might be

```
sort(l) <==  if empty(l)
             then nil
             else merge(head(l)  sort(tail(l))) ,
```

where

```
merge(x l) <== if empty(l)
               then list(x)
               else  if x ≤ head(l)
                     then cons(x l)
                     else cons(head(l)
                               merge(x tail(l))) .
```

The sorting method employed by this program is intrinsically inefficient. The program contains no explicit statement that the list it produces is intended to be ordered. Without such a statement, it is difficult to imagine a system stumbling across a more efficient sorting method.

A more practicable approach would be to have the user specify the purpose of the given program along with the program itself. The system would then apply *correctness–preserving transformations*, which could alter the given program to achieve the same purpose in a fundamentally different way.

> The pure program-transformation approach has been advocated by Burstall and Darlington [1977], Knuth [1974], Standish et al. [1976], and others. Gerhart [1975] introduces a system of correctness-preserving transformations. An experimental system to improve programs by successive transformation was implemented by Darlington and Burstall [1976].

## B. Abstract Data Structures

Out of the different diagnoses of the causes of our programming ills, there arise different therapies. One school of thought attributes much of the difficulty of programming to the process of encoding high-level data structures in terms of the constructs available in the target programming language.

According to this school, we design an algorithm in our minds in terms of *abstract data structures*, structures such as sets, queues, or graphs whose properties are specified but whose precise implementation is undetermined. In these terms, the "mental algorithm" is straightforward and easy to formulate.

The difficulty arises when we attempt to express our mental algorithm in terms of the primitive constructs of the target language, such as arrays or lists. Because the machine representations at our disposal do not correspond precisely to the abstract data structures of our mental algorithm, an act of paraphrase is involved in the programming process. We must simultaneously formulate our algorithm and express it in terms of machine operations. Furthermore, there are often many possible implementations for the same abstract data structure; only after we have completely described our algorithm in abstract terms, and can see what operations are to be performed on the structure, can we decide which implementation will lead to the most efficient program.

It has therefore been proposed that we construct our program in two stages: we begin by constructing a clear program in terms of the abstract data structures of our mental algorithm; only then do we choose a representation for the abstract data structures, and transform our program accordingly. For instance, we would first express our algorithm in terms of high-level operations such as popping an element from a queue or adding an element to a set; then we would decide how to represent the queue or set, as an array or list, say. Facilities might be provided to perform the required transformations automatically, or at least to ensure that they are done correctly.

The transformation process may be regarded as a program-synthesis task. The specification for this task is the program expressed in terms of the abstract data structures; the operations on these structures are considered to be nonprimitive constructs. The properties of the abstract data structures and their operations are stated as transformation rules. The final program will be equivalent to the original, but all the nonprimitive abstract operations will have been reformulated in terms of primitive target-language constructs.

For example, suppose we are writing a program that deals with queues as an abstract data structure. We may have three operations on a queue: a *push* operation, which inserts an element at the end of the queue; a *top* operation, which produces the first element of the queue; and a *pop* operation, which removes the first element from the queue. Informally, we can represent the properties of these operations by the rules

$$push(y\ queue(x_1\ \ldots\ x_n)) \Rightarrow queue(x_1\ \ldots\ x_n\ y)$$

$$top(queue(y\ x_1\ \ldots\ x_n)) \Rightarrow y \quad \text{if } queue(y\ x_1\ \ldots\ x_n) \text{ is nonempty}$$

$$pop(queue(y\ x_1\ \ldots\ x_n)) \Rightarrow queue(x_1\ \ldots\ x_n) \quad \text{if } queue(y\ x_1\ \ldots\ x_n) \text{ is nonempty.}$$

Now, suppose that we have written our program in terms of abstract queues, but that our target programming language requires us to represent our queues in terms of lists. The obvious representation is to encode the queue directly as a list, i.e.,

$$encode1(queue(x_1\ \ldots\ x_n)) \Rightarrow list(x_1\ \ldots\ x_n)\ .$$

An alternate representation is to encode the queue as a list with the elements reversed, i.e.,

$$encode2(queue(x_1\ \ldots\ x_n)) \Rightarrow list(x_n\ \ldots\ x_1)\ .$$

Assume that we have chosen the first encoding.

To our encoding operation *encode*1 there corresponds the opposite decoding operation

$$decode1(list(x_1\ \ldots\ x_n)) \Rightarrow queue(x_1\ \ldots\ x_n)\ .$$

Our synthesis task is now to construct concrete operations on lists that correspond under our chosen encoding to the abstract *push*, *top*, and *pop* operations, i.e.,

```
push1(y l) <== encode1(push(y decode1(l)))

top1(l) <== top(decode1(l))
  where decode1(l) is nonempty

pop1(l) <== encode1(pop(decode1(l)))
  where decode1(l) is nonempty,
```

where *l* is a list. We can consider these descriptions as specifications for a synthesis task in which *push*, *top*, *pop*, *encode*1, and *decode*1 are all regarded as nonprimitive constructs. By including the rules describing the properties of these constructs among our transformation rules, and applying our usual program-synthesis techniques, we obtain the following concrete implementations:

```
push1(y l) <== if empty (l)
               then list(y)
               else cons(head(l)
                             push(y tail(l))) ,

top1(l) <== head(l) ,
```

and

```
pop1(l) <== tail(l) .
```

The final program is then obtained by replacing the abstract operations *push*, *top*, and *pop* by the concrete implementations *push*1, *top*1, and *pop*1 in the given program.

In this implementation, *top*1 and *pop*1 may be executed directly, but *push*1 involves searching down the entire queue. Therefore, we might choose this implementation if the *top* and *pop* operations must be performed quickly, but the *push* operation is permitted to take more time.

If the reverse situation is the case, and *push* is the more critical operation, we may choose the alternate representation, in which the elements of the queue appear on the list in reverse order, i.e.,

$$encode2(queue(x_1 \ldots x_n)) \Rightarrow list(x_n \ldots x_1) .$$

The corresponding implementations that result are

```
push2(y l) <== cons(y l) ,

top2(l) <== if empty(tail(l))
            then head(l)
            else top2(tail(l)) ,
```

and

```
pop2(l) <== if empty(tail(l))
            then nil
            else cons(head(l)
                          pop2(tail(l))) .
```

In this representation, the *push* operation becomes quite economical, but the *top* and *pop* operations become correspondingly more expensive.

The problems that arise in translating abstract data structures into concrete representations require all the synthesis techniques we have considered. However, these problems are of a more limited scope and require less invention than the more general synthesis problem. It is likely that program-synthesis techniques will become practical for such relatively restricted problems long before the general problem is solved.

> The data-abstraction methodology has been investigated extensively (see, for example, Liskov and Zilles [1975] and Guttag, Horowitz, and Musser [1976]). Systems in which the representations for certain abstract data structures are selected automatically have been implemented by Low [1976] and Schwartz [1974]. Our queue example follows Hewitt and Smith [1975], at a safe distance.

## C. Program Modification

It is often remarked that programmers spend more of their time in modifying old programs to achieve additional purposes than in constructing new programs. These modification tasks are conceptually far less challenging than the original programming effort. However, a programmer is especially prone to err in modifying a program: For one thing, if the original program is complex, it may be difficult to find all the points at which changes must be made. Furthermore, the programmer may not know or remember how the program works; he may interfere with its original functioning in introducing the required changes.

Thus, the difficulty of program modification may be attributed to its complexity as a bookkeeping chore rather than to its challenge as a creative endeavor. For this reason, program modification is another area in which program-synthesis techniques are likely to find their earliest application.

We have already introduced a program-modification technique, using protected conditions, as a basis for our simultaneous-goal principle in program synthesis. This technique can also be applied directly to the program-modification task. Thus, we modify the given program to achieve a new condition, while protecting the condition the program was originally intended to achieve.

We have seen one example (in Section 4C) in which our program-modification technique was applied to extend a program for finding the value of the maximum element of an array, to also find the index of that element. The original program,

```
max(a n) <== if $n = 0$
             then $z \leftarrow a[0]$
             else max(a $n-1$)
                  if $z < a[n]$
                  then $z \leftarrow a[n]$ ,
```

was constructed to achieve the condition

$$\mathit{all}(a[0:n]) \le z \mathit{\ and\ } z \in a[0:n] \mathit{\ and\ only\ } z \mathit{\ changed.}$$

This program was then modified to achieve the additional condition

$$z = a[y] \mathit{\ and\ } 0 \le y \le n \mathit{\ \ and\ only\ } y, z \mathit{\ changed}$$

while still maintaining two of the original conditions,

$$\mathit{all}(a[0:n]) \mathit{\ and\ } z \in a[0:n]\ .$$

This modification task was specified as

```
maxindex(a n) <== if $n = 0$
                  then $z \leftarrow a[0]$
                  else maxindex(a $n-1$)
                       if $z < a[n]$
                       then $z \leftarrow a[n]$
                  achieve $a[y] = z$ and $0 \le y \le n$ and only $y, z$ changed
                  protect $all(a[0:n]) \le z$ and $z \in a[0:n]$ .
```

The **achieve** task ensures that the modified program will fulfill its new purpose, and the

**protect** task guarantees that in modifying the program we will not interfere with its original functioning.

From the above specification, we obtained the modified program

```
maxindex(a n) <== if n = 0
                  then y ← 0
                       z ← a[0]
                  else maxindex(a n−1)
                       if z < a[n]
                       then y ← n
                            z ← a[n] .
```

# 6. LOOSE ENDS

## A. A Footnote on Structured Programming

In program synthesis we attempt to reproduce by machine the same process that is carried out by the "structured programmer" by hand. However, the basic programming principles we employ in this paper are not merely machine implementations of the principles of structured programming. Let us briefly examine the derivation of a program in the style of a structured-programming practitioner, to illustrate some of the essential differences.

The program *exp*($x$ $y$) we construct is intended to set the value of the variable $z$ to be the exponential $x^y$ of two integers $x$ and $y$ , where $x$ is positive and $y$ is nonnegative. We assume we are given a number of properties of the exponential function, including

$$u^v = 1 \quad \text{if } u \neq 0 \text{ and } v = 0\ ,$$

$$u^v = (u \cdot u)^{v \div 2} \quad \text{if } v \text{ is even, and}$$

$$u^v = u \cdot (u \cdot u)^{v \div 2} \quad \text{if } v \text{ is odd}\ ,$$

where $u$, $v$, and $w$ are any integers. Here, $\div$ denotes integer division. Written in our notation, the top-level goal of a structured-programming derivation is

**Goal A:** **achieve** $z = x^y$

(where the exponential function $u^v$ is considered to be nonprimitive). This goal can be decomposed into the conjunction of two conditions

**Goal B:** **achieve** $z \cdot xx^{yy} = x^y$ *and* $yy = 0$ .

The motivation given for this step is that, initially, we can achieve the first condition $z \cdot xx^{yy} = x^y$ easily enough (by setting $xx$ to $x$, $yy$ to $y$, and $z$ to 1); if we manage to achieve the second condition $yy = 0$ subsequently, while maintaining the first condition, we will have achieved our goal.

For this purpose, we establish an iterative loop, whose invariant is $z \cdot xx^{yy} = x^y$ and whose exit condition is $yy = 0$; the body of the loop must bring $yy$ closer to zero while maintaining the invariant.

By exploiting the known properties of the exponential and other arithmetic functions, we are led ultimately to a final program, e.g.,

```
exp(x y) <==  (xx yy z) ← (x y 1)
              while yy ≠ 0
              do  if even(yy)
                    then (xx yy) ← (xx·xx  yy÷2)
                    else (xx yy z) ← (xx·xx  yy÷2  xx·z) .
```

The weak point of this derivation seems to be the passage from Goal A to Goal B. This step is necessary to provide the invariant for the loop of the ultimate program. However, how do we know to use this invariant unless we already know the final program in advance? Why should we generate this goal instead of one of the following, equally plausible alternatives;

**Goal B$_1$:** **achieve** $z + xx^{yy} = x^{y}$ *and* $xx = 0$

[to be initialized by $(xx\ yy\ z) \leftarrow (x\ y\ 0)$],

**Goal B$_2$:** **achieve** $z^{yy} = x^{y}$ *and* $yy = 1$

[to be initialized by $(yy\ z) \leftarrow (y\ x)$], or even

**Goal B$_3$:** **achieve** $(z \cdot xx)^{yy} = x^{y}$ *and* $xx = yy = 1$

[to be initialized by $(xx\ yy\ z) \leftarrow (x\ y\ 1)$ or by $(xx\ yy\ z) \leftarrow (1\ y\ x)$] ?

Each of these steps can be motivated by the same considerations that justified the generation of Goal B, but none of them leads to an exponential program so readily.

Our instructors at the Structured Programming School have urged us to find the appropriate invariant assertion before introducing a loop. But how are we to select the successful invariant when there are so many promising candidates around?

The corresponding derivation of the same program by the program-synthesis techniques of this paper requires no such precognitive insights. By using the same properties of the arithmetic functions that were exploited in the structured-programming derivation, we can reduce

**Goal A:** **compute** $x^{y}$

to the two subgoals

**Goal B:** **compute** $(x \cdot x)^{y \div 2}$

(in the case that $y$ is even) and

**Goal C:** **compute** $x \cdot (x \cdot x)^{y \div 2}$

(in the case that $y$ is odd). Only after we observe that the subexpression $(x \cdot x)^{y \div 2}$ , which occurs in both subgoals, is an instance of the expression $x^y$ in the top-level goal, do we actually decide to introduce a recursive call $exp(x \cdot x \;\; y \div 2)$ to compute these subexpressions. The resulting program is

```
exp(x y) <== if y = 0
             then x
             else if even(y)
                  then exp(x·x  y÷2)
                  else x·exp(x·x  y÷2) .
```

This is a recursive version of the previous iterative exponential program, and can actually be transformed into that program by standard recursion-removal techniques.

The recursive calls in the above program arose naturally from the tree of goals in the derivation, and the structure of the final program reflects the structure of that tree. In contrast, the derivation tree for the iterative program had to be forcibly manipulated to induce the invariant to appear.

Recursion seems to be the ideal vehicle for systematic program construction; its use accounts for the relative simplicity of the above derivation. In choosing to emphasize iteration instead, the proponents of structured programming have had to resort to more dubious means.

> The principles of structured programming have been described often in the literature, e.g., by Dahl, Dijkstra, and Hoare [1972], Wirth [1974], and Dijkstra [1976].

## B. Implementation

It is difficult to develop or evaluate heuristic techniques without experimenting with an implementation. The DEDALUS (DEDuctive ALgorithm Ur-Synthesizer) system is a laboratory tool rather than a practical product. The system is implemented in QLISP (Wilber [1976]), an extension of INTERLISP (Teitelman [1974]) that includes pattern-matching and backtracking facilities. In this section, we will describe some of the special characteristics of our implementation without going into very much detail.

The specifications are expressed in a LISP-like notation. Thus, the output specification for the *lessall* program, which we wrote as

$$x < all(l) \,,$$

is represented in the DEDALUS system as

```
(LESS X (ALL L)).
```

The output specification for the *gcd* program, which we wrote as

$$max\{z : z|x \text{ and } z|y\} \,,$$

is represented as

```
(MAX (SETOF Z (AND (DIVIDES Z X)
                   (DIVIDES Z Y)))).
```

The target program is also expressed in LISP syntax.

The transformation rules are expressed as programs in the QLISP programming language. For example, the rule that we denoted by

$$P \text{ and } true \Rightarrow P$$

is represented by the QLISP program

```
(QLAMBDA (AND ←P TRUE) $P).
```

The rule we wrote as

$$u|v \Rightarrow true \qquad \text{if } u \text{ is an integer and } v = 0$$

is expressed as

```
(QLAMBDA (DIVIDE ←U ←V)
         (INSIST (PROVE ('(INTEGER $U))))
         (INSIST (PROVE ('(EQUAL $V 0))))
         TRUE).
```

Although the reader who is unfamiliar with the QLISP language may not understand all the details of the above programs, he may still observe that they are similar in form to the rules that they represent; the features of the QLISP language make this representation fairly direct.

Because rules are represented as programs, we are allowed the full power of the programming language in expressing each rule.

The DEDALUS system currently contains more than a hundred such transformation rules. In expanding the system to handle a new subject domain, we simply introduce new rules.

The rules of the system are classified according to their *pattern*, their left-hand side. This pattern describes the class of subgoals to which the rule can be applied. Thus, the rules

$u|v$ => *true*  if ...

and

$u|v$ => $u|v-u$  if ...

both have pattern $u|v$, and can be applied to goals such as

**compute** $x \mid y+z$ .

When a new goal is generated, the system retrieves those rules whose patterns match the form of the goal. This retrieval is facilitated by arranging the rules in a classification tree according to their patterns; thus the two rules above would be classified on the same branch of the tree. This mechanism allows us to avoid matching every rule in the system against each newly-generated goal.

If no rule matches the entire expression of a goal, its subexpressions are established as subgoals. If no rule matches any subexpression of a given goal, a *failure* occurs, and backtracking is invoked; the system attempts to find an alternate transformation that applies to a previous subgoal.

The QLISP pattern-matcher has special provisions for matching commutative functions. Thus, because the *and* operation is commutative, the rule

*P and true* => *P* ,

represented as the QLISP program

```
(QLAMBDA (AND ←P TRUE) $P),
```

can be applied to goals of form "*true and P*" as well as "*P and true*". For this reason, commutativity rules such as

P *and* Q => Q *and* P

are not necessary in the DEDALUS system.

This kind of matching also occurs in the recursion-formation rule, in determining whether a new goal is an instance of some earlier goal. For example, in the actual synthesis of the *gcd* program, the top-level goal

**compute** $max\{z : z|x \text{ and } z|y\}$

was regarded as an instance of itself with the roles of $x$ and $y$ reversed, because the *and* function is commutative. The recursion-formation rule, therefore, was able to propose the recursive call *gcd*($y$ $x$).

Currently, the DEDALUS implementation incorporates the principles of conditional formation, recursion formation (including the termination proofs), and procedure formation, but not generalization or the formation of structure-changing programs. The techniques for deriving straight-line structure-changing programs were implemented in a separate system (see Waldinger [1977]).

Representative samples of the programs constructed by the current DEDALUS system are the following.

Numerical Programs:
- the subtractive *gcd* algorithm
- the Euclidean *gcd* algorithm
- the binary *gcd* algorithm
- the remainder of dividing two integers

List Programs:
- finding the maximum element of a list
- testing if a list is sorted
- testing if a number is less than every element of a list of numbers (*lessall*)
- testing if every element of one list of numbers is less than every element of another (*allall*)

Set Programs:
- computing the union or intersection of two sets
- testing if an element belongs to a set
- testing if one set is a subset of another
- computing the Cartesian product of two sets (*cart*).

## C. Historical Remarks

In this section we trace briefly the history of the deductive approach to program synthesis.

The early *heuristic compiler* of Simon [1963] constructed simple straight-line list-processing programs from descriptions of the expected input and desired output; the system was based on the General-Problem-Solver approach.

A later group of systems was based on the *resolution theorem-proving approach*: the specifications for the desired program were translated into an equivalent theorem-proving problem, and the desired program was derived from the corresponding proof. (See, e.g., Green [1969], Waldinger and Lee [1969], and Lee, Chang, and Waldinger [1974].) These systems could produce conditional programs, but their loop-formation ability was rudimentary; the required mathematical-induction proofs were awkward to perform in the resolution formalism. Efforts to improve the synthesis of loops within a (nonresolution) theorem-proving approach are described in Manna and Waldinger [1971].

A program-synthesis system based on the program-verification formalism of Hoare [1969] is described by Buchanan and Luckham [1974]. Their system was implemented using some of the facilities of PLANNER (Hewitt [1971]); it required that the loops be specified in advance by the user.

The more recent work in program synthesis is too extensive and too varied to be summarized here. Papers related to aspects of the deductive approach are mentioned in the appropriate sections of the text; some of the other approaches are discussed in the next section.

## D. Other Approaches

The program-synthesis approach we have followed requires that we provide complete specifications for the desired program expressed in an artificial language. It has been argued that these specifications are difficult to provide, and many alternate approaches have been built around different specification schemes.

- *Sample input-output pairs.* In this approach (e.g., see Hardy [1975], Summers [1977]), the program is described by giving typical inputs and the corresponding outputs. Thus,

(A B C) => (C B A) , (A (B C) D) => (D (B C) A)

suggests a program to reverse a list. Such specifications are natural and easy to formulate.

However, in constructing the pairs one must be careful to avoid ambiguities; for instance the pairs

(6 4) => 2 , (13 7) => 6 , (23 13) => 10

could represent either the subtraction or the remainder program. Furthermore, the approach demands that the system be able to generalize from examples, not always an easy task; for instance, it is not immediately obvious that

(2 2) => 4 , (3 6) => 6 , (7 1) => 7 , (14 21) => 42

denotes a least-common-multiple program. Moreover, the generalization task is redundant: the system is trying to guess a relation that the user knows perfectly well, but is unable to express directly in this notation.

- *Sample execution traces.* In this approach, the user provides a detailed trace of the performance of the desired program on some typical inputs. (See, e.g., Biermann and Krishnaswamy [1976].) Thus, the trace

(12 18) → (6 12) → (0 6) → 6

indicates the Euclidean algorithm for the *gcd* function. Here, the possibilities of ambiguity and the burden on the system are reduced, but the user himself is required to design the algorithm to be employed.

- *Predicate-logic language.* This is a direct descendent of the theorem-proving approach. The specifications for the program are expressed as resolution-style clauses; the system then transforms these clauses into another, equivalent set of clauses, which can be regarded as the desired program. (See, e.g., Kowalski [1974], Clark and Sickel [1977].) We question whether the clause form has the notational flexibility to serve as a suitable specification language; for example, many of the constructs we use in our specifications would not usually be permitted in a predicate-logic clause.

- *Synthesis by debugging.* Human programmers produce their programs by the successive debugging of nearly correct programs. It has been proposed that a synthesis system could benefit by imitating this process. In this way, it could focus its attention on the main features of a problem, postponing consideration of the details until afterwards. Such techniques have been applied to the construction of robot plans (Sussman [1975]) and electronic circuits (Sussman [1977]), for example, but not to the solution of more typical programming problems.

- *Synthesis by analogy.* It is unusual for a programmer to construct a program from its specifications by a purely deductive process; normally, he attempts to apply techniques extracted from previous solutions to similar problems. Thus, he might compute the square root of a number by a binary-search technique extracted from a previous program to divide two numbers. Most of the work on this approach (e.g., Manna and Waldinger [1975], Dershowitz and Manna [1977], and Ulrich and Moll [1977]) requires that a close syntactic correspondence be found between the specifications for the two programs; this correspondence then provides a basis for transforming the previous program to solve the new problem. To be more effective, these techniques must be strengthened to take advantage of looser similarities.

- *Automatic programming.* It has been claimed (e.g., see Balzer [1972]) that, for a complex programming task, it is unrealistic to expect the user to formulate complete, correct specifications for the desired program. In specifying an airline-reservation system, an operating system, or a spacecraft-guidance system, for example, we are unlikely to anticipate the desired behavior of the system in every possible situation. In some systems, the specifications for the program are formulated gradually through an extended dialogue between the user and the system. (See, e.g., Green [1976], Barstow [1977], Balzer et al. [1977], or the survey of Heidorn [1976].) The dialogue is continued during the program-construction process, so that the user can resolve any ambiguities or inconsistencies the system might discover. Typically, these systems attempt to play the role of an expert programmer-consultant, and they tend to rely more on built-in knowledge than on deductive processes. By admitting natural language as a communication vehicle, automatic-programming systems avoid the necessity of specifying programs in an artificial formalism; however, they add to the problem of program construction the not inconsiderable difficulties of natural-language understanding.

A survey of various approaches to automatic program construction can be found in Biermann [1976].

## E. Unsettled Questions

Many of the techniques we have presented in this paper bring to mind questions that have not been adequately answered. Some of these are mentioned here.

- **Conditional-formation.** We have introduced a case analysis, and consequently a conditional expression, when we failed in an attempt to prove or disprove some condition. This attempt, however, may be somewhat time-consuming, as it involves exhausting all the rules that might apply to the condition. Moreover, there are certain situations in which we can see in advance that the theorem-proving effort is doomed to failure. For example, if we can find a legitimate input that will cause the condition to be true, and another that will cause the condition to be false, it is clear that we can neither prove nor disprove the condition. Is is possible to recognize some of these situations quickly, thus avoiding the expense of a pointless theorem-proving effort?

- **Generalization.** We formed a generalized procedure when we discovered that two subgoals were an instance of a "somewhat" more general expression. For all the examples in this paper, the only generalizations we require involved replacing a constant by a variable, or replacing one occurrence of a variable by a new variable. In some cases, however, it is necessary to replace a complex term by a new variable. On the other hand, if the specifications for the new procedure are too general, it may be impossible to construct a program that satisfies them. What limits shall we set on the extent of generalization we permit?

- **Termination.** In forming simple recursive programs, it is always possible to establish termination by finding a well-founded ordering between the input of the program and the arguments to its recursive calls. Methods for finding this well-founded ordering during the derivation process have been discovered and implemented in the DEDALUS system. However, we have seen that, to prove the termination of systems of mutually recursive procedures, it is necessary to find termination functions that map all the inputs and arguments into a single well-founded set. How are we to find these termination functions and the related well-founded set during the synthesis process?

- **List-manipulating programs.** We have introduced techniques for forming programs that manipulate data structures. In our examples, however, the only data-structure manipulation we perform is the assignment of values to variables. The same techniques can be applied in a straightforward way to construct array-manipulating programs. Can these techniques be extended to develop programs that change the structure of lists, graphs, and other complex data objects? The in-place list-reversing program and the Schorr-Waite garbage collection algorithm are programs within this category.

- **Simultaneous goals.** The techniques we develop for achieving more than one goal simultaneously presuppose that the transformation rules at our disposal can focus on only one goal at a time, so that the various goals must be achieved, and protection conditions checked, in separate stages. Couldn't we devise transformation rules that, while trying to achieve one condition, consider what conditions have been protected, and what other conditions have yet to be achieved? Thus, a rule that was about to introduce an assignment statement into the program might check whether it is permitted to change the variable.

- **Strategic controls.** We have introduced strategic controls to prevent the derivation tree from growing unmanageably. In the derivation trees constructed by the DEDALUS system, the unsuccessful branches at least represent plausible and well-motivated attempts to solve the problem. Will this mechanism still be adequate when we increase the number of rules from one hundred to one thousand, or the size of the target program from a few lines to a few pages?

- **Efficiency.** The techniques we have introduced are not concerned with the efficiency of the programs they produce. However, if program-synthesis methods are ever to become practical, they must take efficiency considerations into account. This is not to say that a synthesis system will need to perform a mathematical analysis of the program being constructed; it would suffice to find crude estimates of the running time to guide the derivation (cf. Wegbreit [1976], Kant [1977]).

- **Specifications.** The only specifications we have allowed describe the relationships between the expected input and the desired output of the program to be constructed. Such "input-output specifications" are inadequate to describe certain classes of programs. In particular, in specifying, say, an airline-reservation system or an operating system, which are never intended to terminate, it is necessary to express relationships between the inputs it accepts and the outputs it produces at intermediate stages in the computation. Can the techniques we have used with input-output specifications be extended to allow the construction of such "continuously operating programs?"

## Acknowledgements

We would like to acknowledge the value of our discussions with Nachum Dershowitz, and to thank Mike Wilber for help in the use of the QLISP programming system. Don Sannella gave comments on the manuscript.

Earlier versions of sections of this paper have been presented at the Symposium on Artificial Intelligence and Programming Languages, Rochester, NY (August 1977) and the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA (August 1977).

## References

**Aubin, R. [July 1975],** *Some generalization heuristics in proofs by induction,* Proceedings of the IRIA Symposium on Proving and Improving Programs, Arc-et-Senans, France, pp. 197-208.

**Balzer, R. M. [Sept. 1972],** *Automatic programming,* Technical Report, Information Science Institute, University of Southern California, Marina del Rey, CA.

**Balzer, R. M., N. Goldman, and D. Wile [Aug. 1977],** *Informality in program specifications,* Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, pp. 389-397.

**Barstow, D. [Aug. 1977],** *A knowledge-based system for automatic program construction,* Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, pp. 382-388.

**Biermann, A. W. [1976],** *Approaches to automatic programming,* in *Advances in Computers,* Vol. 15, Academic Press, New York, NY, pp. 1-63.

**Biermann, A. W. and R. Krishnaswamy [Sept. 1976],** *Constructing programs from example computations,* IEEE Transactions on Software Engineering, Vol. 2, No. 3, pp. 141-153.

**Bledsoe, W. W. and M. Tyson [1977],** *Typing and proof by cases in program verification,* in *Machine Intelligence* 8: *Machine Representations of Knowledge* (E. W. Elcock and D. Michie, eds.), Ellis Horwood Ltd., Chichester, England, pp. 30-51.

**Boyer, R. S. and J S. Moore [Jan. 1975],** *Proving theorems about LISP functions,* JACM, Vol. 22, No. 1, pp. 129-144.

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

**Brotz, D. [1973],** *Proving theorems by mathematical induction,* Ph.D. thesis, Computer Science Department, Stanford University, Stanford, CA.

**Buchanan, J. R. and D. C. Luckham [May 1974],** *On automating the construction of programs,* Technical Report, Artificial Intelligence Laboratory, Stanford University, Stanford, CA.

**Burstall, R. M. [Feb. 1969],** *Proving properties of programs by structural induction,* Computing J., Vol. 12, No. 1, pp. 41-48.

**Burstall, R. M. and J. Darlington [Jan. 1977],** *A tranformation system for developing recursive programs*, JACM, Vol. 24, No. 1, pp. 44-67.

**Clark, K. and S. Sickel [Aug. 1977],** *Predicate logic: A calculus for deriving programs*, Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, pp. 419-420.

**Dahl, O. J., E. W. Dijkstra, and C. A. R. Hoare [1972],** *Structured programming*, Academic Press, New York, NY.

**Darlington, J. [July 1975],** *Applications of program transformation to program synthesis*, Proceedings of the IRIA Symposium on Proving and Improving Programs, Arc-et-Senans, France, pp. 133-144.

**Darlington, J. and R. M. Burstall [1976],** *A system which automatically improves programs*, Acta Informatica, Vol. 6, No. 1, pp. 41-60.

**Dershowitz, N. and Z. Manna [Nov. 1977],** *The evolution of programs: A system for automatic program modification*, IEEE Transactions on Software Engineering, Vol. SE-3, No. 4.

**Dijkstra, E. W. [Aug. 1975],** *Guarded commands, nondeterminacy and formal derivation of programs*, CACM, Vol. 18, No. 8, pp. 453-457.

**Dijkstra, E. W. [1976],** *A discipline of programming*, Prentice-Hall, Englewood Cliffs, NJ.

**Floyd, R. W. [1967],** *Assigning meanings to programs*, Proceedings of the Symposium in Applied Mathematics, Vol. 19 (J. T. Schwartz, ed.), American Mathematical Society, Providence, RI, pp. 19-32.

**Gerhart, S. L. [Jan. 1975],** *Correctness-preserving program transformations*, Proceedings of the Second Symposium on Principles of Programming Languages, Palo Alto, CA, pp. 54-66.

**Green, C. C. [May 1969],** *Application of theorem proving to problem solving*, Proceedings of the International Joint Conference on Artificial Intelligence, Washington, DC, pp. 219-239.

**Green, C. C. [Oct. 1976],** *The design of the PSI program synthesis system*, Proceedings of the Second International Conference on Software Engineering, San Francisco, CA, pp. 4-18.

**Guttag, J. V., E. Horowitz, and D. R. Musser [Aug. 1976],** *Abstract data types and* *software validation*, Technical Report, Information Sciences Institute, Marina del Rey, CA.

**Hardy, S. [Sept. 1975],** *Synthesis of LISP programs from examples*, Proceedings of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, Georgia, USSR, pp. 240-245.

**Heidorn, G. E. [July 1976],** *Automatic programming through natural language dialogue: A survey*, IBM Journal of Research and Development, Vol. 20, No. 4, pp. 302-313.

**Hewitt, C. [Apr. 1971],** *Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot*, Ph.D. thesis, MIT, Cambridge, MA.

**Hewitt, C. and B. Smith [Mar. 1975],** *Towards a programming apprentice*, IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, pp. 26-45.

**Hoare, C. A. R. [Oct. 1969],** *An axiomatic basis for computer programming*, CACM, Vol. 12, No. 10, pp. 576-580, 583.

**Kant, E. [Aug. 1977],** *The selection of efficient implementations for a high-level language*, Proceedings of the Symposium on Artificial Intelligence and Programming Languages, Rochester, NY, pp. 140-146.

**Knuth, D. E. [Dec. 1974],** *Structured programming with* **go to** *statements*, Computing Surveys, Vol. 6, No. 4, pp. 261-301.

**Kowalski, R. [Aug. 1974],** *Predicate logic as a programming language*, Information Processing 1974, North-Holland, Amsterdam.

**Lee, R. C. T., C. L. Chang, and R. J. Waldinger [Apr. 1974],** *An improved program-synthesizing algorithm and its correctness*, CACM, Vol. 17, No. 4, pp. 211-217.

**Liskov, B. H. and S. N. Zilles [Mar. 1975],** *Specification techniques for data abstractions*, IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, pp. 7-18.

**Low, J. R. [1976],** *Automatic coding: choice of data structures*, Birkhauser Verlag, Basle, Switzerland.

**Luckham, D. C. and J. R. Buchanan [July 1974],** *Automatic generation of programs containing conditional statements*, Proceedings of the Conference on Artificial Intelligence and the Simulation of Behaviour, Sussex, England, pp. 102-126.

**Manna, Z. and R. J. Waldinger [Mar. 1971],** *Toward automatic program synthesis*, CACM, Vol. 14, No. 3, pp. 151-165.

**Manna, Z. and R. J. Waldinger [Summer 1975],** *Knowledge and reasoning in program synthesis*, Artificial Intelligence, Vol. 6, No. 2, pp. 175-208.

**Sacerdoti, E. D. [Sept. 1975],** *The nonlinear nature of plans*, Proceedings of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, USSR, pp. 206-214.

**Schwartz, J. T. [Mar. 1974],** *Automatic and semiautomatic optimization of SETL*, Proceedings of the Symposium on Very High Level Languages, Santa Monica, CA, pp. 43-49.

**Siklossy, L. [1974],** *The synthesis of programs from their properties, and the insane heuristic*, Proceedings of the Third Texas Conference on Computing Systems, Austin, TX.

**Simon, H. A. [Oct. 1963],** *Experiments with a heuristic compiler*, JACM, Vol. 10, No. 4, pp. 493-506. Also in *Representation and Meaning* (H. A. Simon and L. Siklossy, eds.), Prentice-Hall, NJ, 1972.

**Standish, T. A., D. C. Harriman, D. F. Kibler, and J. M. Neighbors [Feb. 1976],** *Improving and refining programs by program manipulation*, Technical Report, University of California, Irvine, CA.

**Summers, P. D. [Jan. 1977],** *A methodology for LISP program construction from examples*, JACM, Vol. 24, No. 1, pp. 161-175.

**Sussman, G. J. [1975],** *A computer model of skill acquisition*, American Elsevier, New York, NY.

**Sussman, G. J. [Aug. 1977],** *Electrical design: A problem for artificial intelligence research*, Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, MA, pp. 894-900.

**Tate, A. [Sept. 1975],** *Interacting goals and their use*, Proceedings of the Fourth International Joint Conference on Artificial Intelligence, Tbilisi, USSR, pp. 215-218.

**Teitelman, W. [1974],** *INTERLISP reference manual*, Xerox Research Center, Palo Alto, CA.

**Ulrich, J. W. and R. Moll [Aug. 1977],** *Program synthesis by analogy*, Proceedings of the Symposium on Artificial Intelligence and Programming Languages, Rochester, NY, pp. 22-28.

**Waldinger, R. J. [1977]**, *Achieving several goals simultaneously*, in *Machine Intelligence* 8: *Machine Representations of Knowledge* (E. W. Elcock and D. Michie, eds.), Ellis Horwood Ltd., Chichester, England, pp. 94-136.

**Waldinger, R. J. and R. C. T. Lee [May 1969]**, *A step toward automatic program writing*, Proceedings of the International Joint Conference on Artificial Intelligence, Washington, DC, pp. 241-252.

**Warren, D. H. D. [June 1974]**, *WARPLAN: A system for generating plans*, Technical Report, Department of Computational Logic, University of Edinburgh, Edinburgh, Scotland.

**Warren, D. H. D. [July 1976]**, *Generating conditional plans and programs*, Proceedings of the Conference on Artificial Intelligence and Simulation of Behaviour, Edinburgh, Scotland, pp. 344-354.

**Wegbreit, B. [Jan. 1976]**, *Goal-directed program transformation*, Proceedings of the Third ACM Symposium on Principles of Programming Languages, Atlanta, GA, pp. 153-170.

**Wilber, B. M. [Mar. 1976]**, *A QLISP reference manual*, Technical Report, SRI International, Menlo Park, CA.

**Wirth, N. [Dec. 1974]**, *On the composition of well-structured programs*, Computing Surveys, Vol. 6, No. 4, pp. 247-259.