ON AUTOMATING STRUCTURED PROGRAMMING

Nachum DERSHOWITZ, Zohar MANNA
Department of Applied Mathematics
The Weizmann Institute of Science, Rehovat (Israël)

ABSTRACT

Structured programming has been advocated in an attempt to impose organization and discipline in the design and development of computer programs. Automating the synthesis of structured programs requires the formalization of the programming techniques involved in order to make them amenable to symbolic manipulation.

We present a number of such techniques and illustrate their applicability in the hand-synthesis of several programs. The programs are developed "top-down" along with their inductive assertions, thereby guaranteeing the correctness of the results. Optimization is touched upon.

Also illustrated is the abstraction of synthesized programs to allow the application of extracted programming techniques in future syntheses.

I. INTRODUCTION

In the last several years, researchers have tried to gain insight into the haphazard art of programming. This has led to the development of "structured programming" which has been defined by Hoare as "the task of organizing one's thought in a way that leads, in a reasonable time, to an understandable expression of a computing task." One of the guidelines of structured programming is that "one should try to develop a program and its proof of correctness hand-in-hand" (Gries [1974]). Much has been written on the subject, including the works of Dijkstra [1968, 1971], Dahl, et al. [1972], Wirth [1971, 1973], Conway and Gries [1973], and others.

The idea is to construct the desired program step by step, beginning with the given input and output specifications. In each step the current goal is solved, transformed to another goal, or reduced to simpler subgoals. Each stage is correct if its predecessor is, thereby guaranteeing the correctness of the final program. Our purpose in this paper is to contribute towards the formalization - and consequently the automatization - of structured programming.

This research is an outgrowth of the recent work by Manna and Waldinger [1975]. We were influenced by Dijkstra's [1973] presentation of his development of the integer square-root function and by the techniques used by Sussman [1973] in his HACKER system.

Our strategies, by themselves, are not expected to lead an automatic program-writing system to the desired solutions of difficult programming problems. Rather, we envision an interactive system (see Floyd [1971]), where the computer takes the more straightforward steps on its own, while the human guides the machine in the more creative ones. Such a system must also have sufficient arithmetic and logical reasoning ability, as that embodied in the verification systems of Deutsch [1973] or Waldinger and Levitt [1974].

One of the major hurdles in this task lies in the formation of loops. Dijkstra [1975], skeptical about some of the claims and goals of "automatic programming", has stated that "while the design of an alternative construct now seems to be a reasonably straightforward activity, that of a repetitive construct requires... 'the invention' of an invariant relation and a variant function." Recent synthesis systems have variously dealt with this problem. Buchanan and Luckham [1974] require the user to supply the skeleton of the loop, and the system fills in the details. Sussman [1973] described his HACKER which, in a limited framework, creates iterative and recursive loops

with no guarantee of correctness. Manna and Waldinger [1975] and others hand-simulated a (partially implemented) synthesizer which can introduce recursion and sometimes strengthen the original specifications. The system described in Green et al. [1974] assumes extensive a priori programming knowledge, such as an experienced programmer would have. Duran [1974] is investigating the use of loop invariants in the synthesis of programs, along lines similar to our loop strategy.

To avoid continually "re-inventing the wheel" we also need learning ability. Just as a human programmer improves with experience - by assimilating various strategies and techniques - so should a working program-writing system learn from old programs, store their more "interesting" aspects, and then judiciously apply them to new problems. We therefore illustrate the abstraction of program segments, thereby obtaining program schemas along with sufficient conditions for their correct application. The use of such schemas, which may represent general subroutines or important programming techniques, is also demonstrated.

Sussman [1973] recognized the importance of programming-skill acquisition. However, since his system relies on debugging rather than formal verification, our results are different. Gerhart [1975] suggests the compilation of a handbook of schemas similar to those we abstract. The modification of an already existing program to solve a somewhat different problem has been found to be a powerful approach by Manna and Waldinger [1975].

In Section II we introduce the programming strategies, and in the following section they are employed in the hand-syntheses of several programs. Our first example is a straightforward synthesis of the integer square-root function. Arrays are introduced in the second example, which is a program to find a maximal element of an array. We conclude with an ambitious attempt at synthesizing Hoare's FIND [1961, 1971] algorithm. Section IV is devoted to abstraction.

II. STRATEGIES

We outline in this section a number of general programming strategies, which have been found useful. Examples of their use may be found in the next section. We do not, however, present detailed heuristics for guiding the choice of strategy.

In the following, the p's represent arbitrary predicates, the t's terms, the y's and z's are variables, while the x's may be either variables or constants.

A goal or subgoal consists of input and output specifications in mathematical logic. We use the general form

assert $p_1(\bar{x})$
achieve $p_2(\bar{x},\bar{y})$

where the predicate $p_1(\bar{x})$ expresses the given relationship between the input variables, and $p_2(\bar{x},\bar{y})$ expresses the desired relationship between the input and program variables upon termination.

Our aim is to transform the unrealized achieve statement into an annotated program segment

assert $p_1(\bar{x})$
purpose $p_2(\bar{x},\bar{y})$
< program segment >
assert $p_2(\bar{x},\bar{y})$ eop

An assert statement such as "assert $p_2(\bar{x},\bar{y})$" corresponds to an inductive assertion (Floyd [1967] and Hoare [1969]; see also Manna [1974]), and indicates that whenever control reaches the statement, for the current values of the variables $\bar{x}$ and $\bar{y}$, the predicate $p_2(\bar{x},\bar{y})$ is true. While constructing the program segment to achieve $p_2(\bar{x},\bar{y})$, the relation $p_1(\bar{x})$ is assumed to hold. The purpose statement is a comment which precedes a program segment and indicates what that segment is meant to achieve. The end of its scope is indicated by eop (end of purpose).

1. Transformations.

a. Equivalence transformations. Any expression may be replaced by an equivalent expression. That is, a predicate $p_1(\bar{x})$ may be replaced by $p_2(\bar{x})$, if we know (or can prove) $p_1(\bar{x}) \equiv p_2(\bar{x})$. Similarly, a term $t_1(\bar{x})$ may be replaced by $t_2(\bar{x})$, if $t_1(\bar{x}) = t_2(\bar{x})$.

b. Strengthening transformations. Any logical expression to be achieved may be replaced by a stronger expression. That is, if $p_2(\bar{x}) \supset p_1(\bar{x})$, then

achieve $p_1(\bar{x})$

may be replaced by

purpose $p_1(\bar{x})$
achieve $p_2(\bar{x})$
assert $p_2(\bar{x})$ eop

We make particular use of the following transformations:

(i) $p(\bar{t}(\bar{x},\bar{z}))$ becomes $p(\bar{t}(\bar{y},\bar{z})) \wedge \bar{t}(\bar{y},\bar{z}) = \bar{t}(\bar{x},\bar{z})$,

(ii) $p(\bar{t}(\bar{x},\bar{z}))$ becomes $p(\bar{t}(\bar{y},\bar{z})) \wedge \bar{y} = \bar{x}$,

(iii) $p(\bar{t}(\bar{x},\bar{z}))$ becomes $p(\bar{y}) \wedge \bar{y} = \bar{t}(\bar{x},\bar{z})$.

Here, $\bar{t}(\bar{x},\bar{z})$ stands for a vector of terms $(t_1(\bar{x},\bar{z}), t_2(\bar{x},\bar{z}),\ldots, t_n(\bar{x},\bar{z}))$ and $\bar{t}(\bar{x},\bar{z})=\bar{t}(\bar{y},\bar{z})$ means $t_1(\bar{x},\bar{z})=t_1(\bar{y},\bar{z}) \wedge t_2(\bar{x},\bar{z})=t_2(\bar{y},\bar{z}) \wedge \ldots \wedge t_n(\bar{x},\bar{z})=t_n(\bar{y},\bar{z})$.

E.g., (i) $z=\gcd(x_1,x_2)$ becomes $z=\gcd(y_1,y_2) \wedge \gcd(y_1,y_2)=\gcd(x_1,x_2)$,

(ii) $(\forall k)(1\leq k\leq n)(p(k))$ becomes $(\forall k)(1\leq k\leq y)(p(k)) \wedge y=n$,

(iii) $z_1=z_2+\sum_k f(k)$ becomes $z_1=z_2+y \wedge y=\sum_k f(k)$.

[For convenience, we separate the range part of a quantified expression: $(\forall k)(r(k))(p(k))$ is $(\forall k)(r(k)\supset p(k))$ and $(\exists k)(r(k))(p(k))$ is $(\exists k)(r(k)\wedge p(k))$.] Notice that new program variables $\bar{y}$ are introduced by these transformations. Transformations (i) and (ii) split a goal into two conjuncts, possibly for use by the loop-until-repeat strategy below. Transformation (iii) introduces program variables equal to terms appearing in the goal. This allows the saving of previous computations during loop execution.

Specific instances of these transformations are used in the examples of the next section, and are similar to the "top-down" heuristics found useful by Katz and Manna [1973] and Wegbreit [1974] in their investigation of the automatic derivation of inductive assertions for program verification.

2. Assignments.

For the purposes of the examples presented in the next section, the following is sufficient. Reduce the conjunctive goal

achieve $p(\bar{x})$, $\bar{y}=\bar{t}(\bar{x})$

that is, "achieve $p(\bar{x}) \wedge y_1=t_1(\bar{x}) \wedge y_2=t_2(\bar{x}) \wedge \ldots \wedge y_n=t_n(\bar{x})$", to

purpose $p(\bar{x})$, $\bar{y}=\bar{t}(\bar{x})$ achieve $p(\bar{x})$ $\bar{y} \leftarrow \bar{t}(\bar{x})$ assert $p(\bar{x})$, $\bar{y}=\bar{t}(\bar{x})$ eop

where the y variables are distinct from the x variables and each $t_j(\bar{x})$ is composed only of primitive operations. The second conjunct $\bar{y}=\bar{t}(\bar{x})$ of the achieve gives rise to the multiple assignment $\bar{y}\leftarrow\bar{t}(\bar{x})$, (i.e., the simultaneous assignment of $t_1(\bar{x})$ to y_1 , $t_2(\bar{x})$ to y_2 , etc.).

3. Protection and splitting.

a. Protection. Given a goal of the form

assert $p_1(\bar{y})$ achieve $p_1(\bar{y}')$, $p_2(\bar{y}',\bar{z})$

[The $\bar{y}'$ are the new values of $\bar{y}$. Primed variables shall be used whenever needed to differentiate between old and new values.] It is reasonable to try to "protect" the relation p_1 from "clobbering" by keeping $\bar{y}$ constant, that is,

```
assert p1(ȳ)
purpose p1(ȳ') , p2(ȳ',z̄)
achieve ȳ'=ȳ , p2(ȳ,z̄)
assert p1(ȳ) , p2(ȳ,z̄) eop
```

b. Splitting. Similarly, given the conjunctive goal

```
achieve p1(ȳ) , p2(ȳ,z̄)
```

(where $\bar{y}$ and $\bar{z}$ are distinct variables), try separating the combined goal into two consecutive subgoals (cf. Sussman's [1973] linear AND-technique and protection mechanism):

```
purpose p1(ȳ) , p2(ȳ,z̄)
achieve p1(ȳ)
achieve ȳ'=ȳ , p2(ȳ,z̄)
assert p1(ȳ) , p2(ȳ,z̄) eop
```

In other words, first achieve p_1, and then hold the variables appearing in p_1 constant - thereby preserving p_1 - while achieving p_2 by setting the $\bar{z}$ variables.

4. The if-then-else-fi strategy.

A goal

```
achieve p1(ȳ,z̄)
```

may be transformed into an equivalent disjunctive goal

```
purpose p1(ȳ,z̄)
   achieve p2(ȳ) , p3(ȳ,z̄)
or achieve p1(ȳ,z̄)
assert [p2(ȳ)∧p3(ȳ,z̄)] ∨ p1(ȳ,z̄) eop
```

if $p_2(\bar{y})\wedge p_3(\bar{y},\bar{z}) \supset p_1(\bar{y},\bar{z})$. There may exist cases where the first disjunct is more directly achievable than the original goal.

If the predicate $p_2(\bar{y})$ is composed only of primitive operators, then try forming the conditional statement:

```
purpose p1(ȳ,z̄)
if p2(ȳ) then assert p2(ȳ)
              achieve ȳ'=ȳ , p3(ȳ,z̄)
         else assert ~p2(ȳ)
                 achieve p2(ȳ') , p3(ȳ',z̄)
              or achieve p1(ȳ',z̄)
         fi
assert [p2(ȳ)∧p3(ȳ,z̄)] ∨ p1(ȳ,z̄) eop
```

The conditional statement contains two subgoals. In the first, p_2 has been achieved (by testing for it) and only p_3 remains; while in the second, the knowledge that $\sim p_2(\bar{y})$ may be used in achieving the subgoal.

5. The loop-until-repeat strategy.*/

For a conjunctive goal

```
achieve p1(x̄,ȳ) , p2(x̄,ȳ)
```

try (if simpler strategies fail)

```
purpose p1(x̄,ȳ) , p2(x̄,ȳ)
achieve p1(x̄,ȳ)
loop assert p1(x̄,ȳ)
     until p2(x̄,ȳ)
     assert p1(x̄,ȳ) , ~p2(x̄,ȳ)
     achieve p1(x̄,ȳ') , ȳ'≠ȳ
     repeat
assert p1(x̄,ȳ) , p2(x̄,ȳ) eop
```

This is the "top-down" strategy where a conjunctive goal $p_1(\bar{x},\bar{y})\wedge p_2(\bar{x},\bar{y})$ is achieved by creating a loop in which one conjunct, $p_1(\bar{x},\bar{y})$, remains invariant - i.e., it is asserted true for the initial values of $\bar{y}$ and for subsequent values of $\bar{y}$ whenever the loop is repeated - until the second conjunct, $p_2(\bar{x},\bar{y})$, is found true. If the loop is not exited, $p_2(\bar{x},\bar{y})$ must be false and the invariant is re-achieved.

The conjunct $\bar{y}'\neq\bar{y}$ (i.e., $y_1'\neq y_1 \vee y_2'\neq y_2 \vee \ldots \vee y_n'\neq y_n$) is meant to avoid achieving $p_1(\bar{x},\bar{y}')$ by letting the new values $\bar{y}'$ equal the old values $\bar{y}$. (It is often left implicit in the following.) Clearly $\bar{y}'\neq\bar{y}$ is in itself insufficient for guaranteeing the

*/ The loop-until-repeat statement is a single-exit loop construct, enclosed by loop and repeat. It is exited when control first reaches the until clause and the exit condition holds.

termination of the loop. What is needed is a "significant" advance in $\bar{y}$ towards the exit condition $p_2(\bar{x},\bar{y})$. For this purpose, we use the range strategy below.

There is a measure of freedom in dividing the goal into an invariant and test. Usually, the "stronger" the invariant and "weaker" the test, the more efficient the resulting loop. Often, a loop may be improved by backtracking and adding to the loop invariant conjuncts such as $\hat{y}=t(\bar{x},\bar{y})$ for terms $t(\bar{x},\bar{y})$ that either appear often within the loop or are relatively difficult to compute. Alternatively, we might discover the need for $\hat{y}$ while synthesizing the loop body.

6. The range strategy.

Given a partially synthesized loop:

```
purpose p1(x̄,ȳ) , p2(x̄,ȳ)
assert p1(x̄,ȳ) , p3(x̄,ȳ)
loop assert p1(x̄,ȳ)
     until p2(x̄,ȳ)
     assert p1(x̄,ȳ) , ~p2(x̄,ȳ)
     achieve p1(x̄,ȳ') , ȳ'≠ȳ
     repeat
assert p1(x̄,ȳ) , p2(x̄,ȳ)
```

with the unrealized subgoal

$$\underline{\text{achieve}}\ p_1(\bar{x},\bar{y}')\ ,\ \bar{y}'\neq\bar{y}\ .$$

First hypothesize the direction of change for some of the integer (or real) variables y_j, and then, if possible, ascertain bounds for those y_j.

a. Direction. Let $\bar{y}^0$ denote the initial values of $\bar{y}$ upon entering the loop, and $\bar{y}^f$ the final values when the loop is exited. Therefore assume $p_1(\bar{x},\bar{y}^0)$, $p_3(\bar{x},\bar{y}^0)$, $p_1(\bar{x},\bar{y}^f)$ and $p_2(\bar{x},\bar{y}^f)$ and attempt to prove for some program variable y_j, either

(i) $y_j^0 \leq y_j^f$, or

(ii) $y_j^f \leq y_j^0$.

Ignoring termination for the moment, what the range strategy suggests is to assume in case (i) that y_j is monotonically increasing during loop execution, i.e., $y_j^0 \leq y_j \leq y_j' \leq y_j^f$, and therefore

$$\underline{\text{achieve}}\ p_1(\bar{x},\bar{y}')\ ,\ \bar{y}'\neq\bar{y}\ ,\ y_j\leq y_j'\ .$$

On the other hand, in case (ii) we assume that y_j is monotonically decreasing during loop execution, i.e., $y_j^f \leq y_j' \leq y_j \leq y_j^0$, and

achieve $p_1(\bar{x},\bar{y}')$, $\bar{y}'\neq\bar{y}$, $y_j'\leq y_j$.

b. Bounds. Note that the range strategy may be applicable to more than one variable y_j . After hypothesizing direction for all those variables, we try to find explicit lower bounds $\ell_j(\bar{x},\bar{y})$ and/or upper bounds $u_j(\bar{x},\bar{y})$ for each y_j . (Obviously $\ell_j(\bar{x},\bar{y})$ and $u_j(\bar{x},\bar{y})$ should not contain y_j itself.) If successful, these relations $\ell_j(\bar{x},\bar{y})\leq y_j$ and/or $y_j\leq u_j(\bar{x},\bar{y})$ may serve as useful loop invariants.

To summarize, for each y_j we may add conjuncts, thus obtaining:

assert $p_1(\bar{x},\bar{y})$, $p_3(\bar{x},\bar{y})$, $\ell_j(\bar{x},\bar{y})\leq y_j$, $y_j\leq u_j(\bar{x},\bar{y})$
loop assert $p_1(\bar{x},\bar{y})$, $\ell_j(\bar{x},\bar{y})\leq y_j$, $y_j\leq u_j(\bar{x},\bar{y})$
  until $p_2(\bar{x},\bar{y})$
  assert $p_1(\bar{x},\bar{y})$, $\sim p_2(\bar{x},\bar{y})$, $\ell_j(\bar{x},\bar{y})\leq y_j$, $y_j\leq u_j(\bar{x},\bar{y})$
  achieve $p_1(\bar{x},\bar{y}')$, $\bar{y}'\neq\bar{y}$, $\ell_j(\bar{x},\bar{y}')\leq y_j'$, $y_j'\leq u_j(\bar{x},\bar{y}')$,
    { $y_j\leq y_j'$ in case (i)
    { $y_j'\leq y_j$ in case (ii)
  repeat
assert $p_1(\bar{x},\bar{y})$, $p_2(\bar{x},\bar{y})$, $\ell_j(\bar{x},\bar{y})\leq y_j$, $y_j\leq u_j(\bar{x},\bar{y})$

Note that this strategy only suggests adding $y_j'\leq y_j$ (or $y_j\leq y_j'$), $\ell_j(\bar{x},\bar{y})\leq y_j$, and $y_j\leq u_j(\bar{x},\bar{y})$. Though the range strategy is often helpful and should be tried, in some cases the assumption of monotonicity may not lead to a solution.

Returning to termination, consider, as an example, the case where the strategy suggests increasing y_1 and decreasing y_2 and we have succeeded in finding bounds u_1 and ℓ_2 . The goal, then, is of the form

assert $p_1(\bar{x},\bar{y})$, $\sim p_2(\bar{x},\bar{y})$, $y_1\leq u_1(\bar{x},\bar{y})$, $\ell_2(\bar{x},\bar{y})\leq y_2$
achieve $p_1(\bar{x},\bar{y}')$, $(y_1',y_2')\neq(y_1,y_2)$, $y_1\leq y_1'\leq u_1(\bar{x},\bar{y}')$,
  $\ell_2(\bar{x},\bar{y}')\leq y_2'\leq y_2$.

In situations such as this, it is often valuable to strengthen the requirement $\bar{y}'\neq\bar{y}$ by limiting it to a subset of $\bar{y}$, here the bounded variables $\{y_1,y_2\}$. Equivalently, we have

achieve $p_1(\bar{x},\bar{y}')$, $y_1<y_1'\leq u_1(\bar{x},\bar{y}')$, $y_2'=y_2$
or achieve $p_1(\bar{x},\bar{y}')$, $y_1'=y_1$, $\ell_2(\bar{x},\bar{y}')\leq y_2'<y_2$
or achieve $p_1(\bar{x},\bar{y}')$, $y_1<y_1'\leq u_1(\bar{x},\bar{y}')$, $\ell_2(\bar{x},\bar{y}')\leq y_2'<y_2$.

Since we have the upper bound $u_1(\bar{x},\bar{y}')$ for the increasing y_1 and lower bound $\ell_2(\bar{x},\bar{y}')$ for the decreasing y_2, if both y_1 and y_2 are integers, then termination of this loop is guaranteed. In all the examples presented below, loop termination shall be guaranteed by such a strict increase/decrease in at least one of the bounded integers.

III. EXAMPLES

In this section we describe the hand-synthesis of several programs, with reference to the strategies of the previous section. Only the successful path of each synthesis is shown, though obviously an implemented system would take up false leads - or synthesize alternative programs - before backtracking and developing the programs as presented.

It is assumed that we have all the necessary logical and arithmetic knowledge along with domain-dependent knowledge to perform the syntheses. Such information shall be introduced when needed as fact statements. For example,

fact $(u \lessdot \sqrt{v}) \equiv (u^2 \lessdot v)$ where $u \geqslant 0 \wedge$ integer(u) */ .

Example 1

We begin with the synthesis of the integer square-root function. Our goal is to synthesize the program:

1.1
```
begin constant integer x
      assert x≥0
      variable integer z
      achieve z=⌊√x⌋
      end
```

In other words, we must construct a program which, for all integers $x \geqslant 0$, computes $\lfloor\sqrt{x}\rfloor$ (i.e., the largest integer less than or equal to the square-root of x). Note that x is a constant whose value may not be changed by the program, while z is a program variable which upon termination must have the value $\lfloor\sqrt{x}\rfloor$.

The top-level goal is

achieve $z=\lfloor\sqrt{x}\rfloor$.

*/ This fact is true for any expressions u or v. In this and other facts, u, v, w, etc. are considered to be universally quantified. "fact $p_1(\bar{u})$ where $p_2(\bar{u})$" means "fact $(\forall\bar{u})[p_2(\bar{u}) \supset p_1(\bar{u})]$".

Using facts about the floor and square-root functions, it may be transformed into the equivalent goal

```
purpose z=⌊√x⌋
achieve z²≤x , x<(z+1)² eop
```

This is a conjunctive subgoal, and since we do not succeed in directly achieving both conjuncts, we try the loop-until-repeat strategy. The conjunct $z^2 \leq x$ is chosen for the loop invariant, and $x<(z+1)^2$ for the exit test.

1.2
```
begin assert x≥0
      achieve z²≤x
      loop assert z²≤x
           until x<(z+1)²
           assert z²≤x , x≥(z+1)²
           achieve z'²≤x , z'≠z
           repeat
      assert z=⌊√x⌋
      end
```

To initialize the loop invariant, we must

```
achieve z²≤x .
```

Since we have asserted that $0 \leq x$, and inequality is transitive, i.e.,

```
fact u≤v ⊃ u≤w where v≤w ,
```

it suffices - by the strengthening transformation (letting $u=z^2$, $v=0$ and $w=x$) - to

```
achieve z²≤0 .
```

Our knowledge of squares allows this to be replaced by

```
achieve z=0 ,
```

which is readily achieved by the assignment:

```
purpose z²≤x
z ← 0
assert z=0 eop
```

Before synthesizing the loop body, we would like to find the relation between z upon entrance to the loop and upon exit, and then apply the range strategy. Assuming $x \geq 0$, $z^0=0$ upon entrance, and $(z^f)^2 \leq x < (z^f+1)^2$ upon exit, we can derive $z^0=0 \leq z^f \leq \sqrt{x}$, using

facts about square-roots. This is case (i) of the range strategy, which suggests increasing z in the loop body, with a lower bound 0 and upper bound $\sqrt{x}$ for z.

1.3
```
begin assert x≥0
      z ← 0
      assert z=0
      loop assert z²≤x , 0≤z≤√x
           until x<(z+1)²
           assert z²≤x , x≥(z+1)² , 0≤z≤√x
           achieve z'²≤x , z'≠z , 0≤z'≤√x , z≤z'
           repeat
      end
```

We now take up the loop body subgoal

assert $z^2 \leq x$, $x \geq (z+1)^2$, $0 \leq z \leq \sqrt{x}$
achieve $z'^2 \leq x$, $z < z' \leq \sqrt{x}$.

Termination is assured since z is an increasing integer, bounded from above. The conjunct $z' \leq \sqrt{x}$ can be dropped since it is implied by $z'^2 \leq x$. We assert $(z+1)^2 \leq x$ and are looking for a z' such that $z'^2 \leq x$, so by the transitivity of inequality it is sufficient to

achieve $z'^2 \leq (z+1)^2$, $z < z'$

or (eliminating the square, since $z' > z \geq 0$)

achieve $z < z' \leq z+1$.

This is achieved by the assignment

```
purpose z'²≤x , z<z'≤√x
z ← z+1 eop
```

The complete annotated program is

1.4
```
begin assert x≥0
      purpose z²≤x , x<(z+1)² , 0≤z≤√x
      purpose z²≤x , 0≤z≤√x
      z ← 0
      assert z=0 eop
      loop assert z²≤x , 0≤z≤√x
           until x<(z+1)²
           assert z²≤x , 0≤z≤√x , x≥(z+1)²
```

```
        purpose z'^2≤x , z<z'≤√x
        z ← z+1 eop
        repeat
    assert z=⌊√x⌋ eop
    end
```

We now wish to optimize this program. The exit test $x<(z+1)^2$ is relatively difficult to compute since it involves multiplication. Accordingly, we wish to replace the exit test with $x<y_1$ and add the invariant $y_1=(z+1)^2$ throughout the loop. The loop initialization had the

$$\underline{\text{purpose}}\ z^2\leq x\ ,\ 0\leq z\leq\sqrt{x}\ .$$

(the conjunct $0\leq z\leq\sqrt{x}$ was added by the range strategy), and now we wish to

$$\underline{\text{achieve}}\ z^2\leq x\ ,\ 0\leq z\leq\sqrt{x}\ ,\ y_1=(z+1)^2\ .$$

The first two conjuncts were solved by setting z to 0, so we are left with

$$\underline{\text{achieve}}\ z=0\ ,\ y_1=(z+1)^2\ .$$

Our new initialization is therefore

$$\boxed{(z,y_1)\leftarrow(0,1)}$$

We now re-solve the loop-body subgoal

$$\underline{\text{achieve}}\ z'^2\leq x\ ,\ z<z'\leq\sqrt{x}\ ,\ y_1'=(z'+1)^2\ .$$

This becomes,

$$\underline{\text{achieve}}\ z'=z+1\ ,\ y_1'=(z'+1)^2$$

or by eliminating z' from the expression for y_1' and expanding the square (in order to extract the old value of y_1)

$$\underline{\text{achieve}}\ z'=z+1\ ,\ y_1'=z^2+4z+4\ .$$

Having asserted that $y_1=(z+1)^2=z^2+2z+1$, we can set

$$\boxed{(z,y_1)\leftarrow(z+1,y_1+2z+3)}$$

Similarly, if we wish to optimize the assignment $y_1\leftarrow y_1+2z+3$, we can also keep $y_2=2z+3$ invariant. Then, following the same procedure as above, we finally obtain the program:

1.5
```
begin constant integer x
      assert x≥0
      variable integer z,y1,y2
      purpose z²≤x , x<(z+1)² , 0≤z≤√x , y1=(z+1)² , y2=2z+3
      (z,y1,y2) ← (0,1,3)
      loop assert z²≤x , 0≤z≤√x , y1=(z+1)² , y2=2z+3
           until x<y1
           purpose z'²≤x , z<z'≤√x , y1'=(z'+1)² , y2'=2z'+3
           (z,y1,y2) ← (z+1,y1+y2,y2+2) eop
           repeat eop
      assert z=⌊√x⌋
      end
```

Example 2

We synthesize here a search for the maximum of an array:

2.1
```
begin constant integer n
      assert 1≤n
      constant real array X[1:n]
      variable integer z
      achieve X[z]≥X[1:n] , 1≤z≤n
      end
```

where $X[z]\geq X[i:j]$ is short for $(\forall k)(i\leq k\leq j)(X[z]\geq X[k])$. Our goal then is to

$$\underline{\text{achieve}}\ X[z]\geq X[1:n]\ ,\ 1\leq z\leq n\ .$$

Note that X is constant, so only the value of z may be changed. Strengthening, by transforming the implicit range, gives

$$\underline{\text{achieve}}\ X[z]\geq X[1:y_1]\ ,\ 1\leq z\leq n\ ,\ y_1=n$$

which is amenable to the loop-until-repeat strategy. (We usually prefer to transform the range part of a quantified expression. Note that $y_1\geq n$ is sufficient for $X[z]\geq X[1:y_1]$ to imply $X[z]\geq X[1:n]$.)

2.2
```
achieve X[z]≥X[1:y1] , 1≤z≤n
loop assert X[z]≥X[1:y1] , 1≤z≤n
     until y1=n
     assert X[z]≥X[1:y1] , 1≤z≤n , y1≠n
     achieve X[z']≥X[1:y1'] , 1≤z'≤n
     repeat
```

Using

fact $A[u:u]=A[u]$

in order to eliminate the quantifier in $X[1:y_1]$ and matching u with y_1 and 1, the initialization may be replaced by

achieve $X[z]\geq X[1]$, $y_1=1$, $1\leq z\leq n$.

The first conjunct can obviously be achieved by $z=1$, so we assign

$(z,y_1) \leftarrow (1,1)$

We now try the range strategy. Obviously $y_1^0=1 \leq n=y_1^f$, and we therefore add the invariant $1\leq y_1\leq n$ throughout the loop, and increment y_1 within the loop body.

$(z,y_1) \leftarrow (1,1)$
loop assert $X[z]\geq X[1:y_1]$, $1\leq z\leq n$, $1\leq y_1\leq n$
  until $y_1=n$
  assert $X[z]\geq X[1:y_1]$, $1\leq z\leq n$, $y_1\neq n$, $1\leq y_1\leq n$
  achieve $X[z']\geq X[1:y_1']$, $1\leq z'\leq n$, $1\leq y_1'\leq n$, $y_1<y_1'$
  repeat

Since the integer y_1 is bound from above and we require $y_1<y_1'$ in the loop body, termination is guaranteed. (Note that the range strategy could have also been applied to z.) We are trying to solve

assert $X[z]\geq X[1:y_1]$, $1\leq z\leq n$, $1\leq y_1<n$
achieve $X[z']\geq X[1:y_1']$, $1\leq z'\leq n$, $y_1<y_1'\leq n$,

and we use the

fact $(p(A[\ell:m]) \wedge p(A[m+1:u])) \equiv p(A[\ell:u])$ where $\ell-1\leq m\leq u$

to split the range of the implied quantifier $[1:y_1']$ into what has already been achieved $([1:y_1])$ and what has yet to be achieved $([y_1+1:y_1'])$. We get

achieve $X[z']\geq X[1:y_1]$, $X[z']\geq X[y_1+1:y_1']$, $1\leq z'\leq n$, $y_1<y_1'\leq n$

In order to simplify $X[y_1+1:y_1']$, we take $y_1'=y_1+1$ and obtain the subgoal

achieve $X[z']\geq X[1:y_1]$, $X[z']\geq X[y_1+1]$, $1\leq z'\leq n$, $y_1'=y_1+1$,
  $y_1<y_1'\leq n$,

which may be reduced to

achieve $X[z']\geq X[1:y_1]$, $X[z']\geq X[y_1+1]$, $1\leq z'\leq n$
$y_1 \leftarrow y_1+1$

We have asserted $X[z]\geq X[1{:}y_1]$ and wish to achieve $X[z']\geq X[1{:}y_1]$, so we break the goal into two disjuncts, protecting the relation in the first

```
assert X[z]≥X[1:y1] , 1≤z≤n , 1≤y1<n
   achieve X[z]≥X[y1+1] , 1≤z≤n , z'=z
or achieve X[z']≥X[1:y1] , X[z']≥X[y1+1] , 1≤z'≤n .
```

Using the if-then-else-fi strategy, we now construct a conditional statement and test for $X[z]\geq X[y_1+1]$.

```
purpose X[z']≥X[1:y1] , X[z']≥X[y1+1] , 1≤z'≤n
if X[z]≥X[y1+1] then assert X[z]≥X[1:y1] , 1≤z≤n , 1≤y1<n ,
                              X[z]≥X[y1+1]
                      achieve z'=z
                else assert X[z]≥X[1:y1] , 1≤z≤n , 1≤y1<n ,
                              X[z]<X[y1+1]
                      achieve X[z']≥X[1:y1] , X[z']≥X[y1+1]
                              1≤z'≤n
                fi eop
```

The first subgoal is trivially achieved by the null statement. For the second subgoal we assume $X[z]<X[y_1+1]$ and thereby know $X[y_1+1]>X[z]\geq X[1{:}y_1]$ and $X[y_1+1]\geq X[y_1+1]$. So the second subgoal may be achieved by $z'=y_1+1$.

All together, we have synthesized

2.4

```
begin constant integer n
      assert 1≤n
      constant real array X[1:n] ; variable integer z,y1
      purpose X[z]≥X[1:n] , 1≤z≤n , y1=n
      (z,y1) ← (1,1)
      loop assert X[z]≥X[1:y1] , 1≤z≤n , 1≤y1≤n
           until y1=n
           purpose X[z']≥X[1:y1'] , X[z']≥X[y1+1] , 1≤z'≤n
           if X[z]≥X[y1+1] then null else z ← y1+1 fi eop
           purpose X[z']≥X[1:y1'] , 1≤z'≤n , y1'=y1+1 , y1<y1'≤n
           y1 ← y1+1 eop
           repeat eop
      assert X[z]≥X[1:n] , 1≤z≤n
      end
```

At this point it is possible to optimize as in Example 1, e.g., by keeping $y_2=X[z]$ invariant.

Example 3

In this last example we illustrate a "top-down" development of the FIND algorithm (Hoare [1961, 1971], see also Dijkstra [1971]). This being a lengthy example, we shall leave out details of the more obvious steps.

The problem is to rearrange the array A , so that A[f] is f-th in order of magnitude; all elements to the left of f have lesser values, and those to the right have greater values. Thus, we begin with

3.1
```
begin constant integer f,n
      assert 1≤f≤n
      variable real array A[1:n]
      achieve A[1:f-1]≤A[f]≤A[f+1:n]
      end
```

We understand A[1:f-1]≤A[f]≤A[f+1:n] to be equivalent to

$$(\forall \ell)(1\leq\ell<f-1)(A[\ell]\leq A[f]) \wedge (\forall \ell)(f+1<\ell\leq n)(A[f]\leq A[\ell]) \ .$$

Note that the output specification should also include the requirement that the final value of the array A be a permutation of the original. This shall be achieved by using the operation exchange(A[i],A[j]) which has the effect:

$$A'[i]=A[j] \wedge A'[j]=A[i] \wedge (\forall k)(k\neq i \wedge k\neq j)(A'[k]=A[k]) \ .$$

Our goal is to

achieve A[1:f-1]≤A[f]≤A[f+1:n] .

This could be developed into a (partial) sort program. However, since we are looking for a more efficient solution, we introduce a quantifier and obtain the equivalent goal

achieve A[1:f-1]≤A[f:f]≤A[f+1:n] .

Now we transform the inner range, keeping the left side adjacent to the lower bound, and the right side adjacent to the upper bound. So

achieve A[1:i-1]≤A[i:j]≤A[j+1:n] , i=f=j .

What we have done is to divide A into three ordered segments A[1:i-1] , A[i:j] and A[j+1:n] and we shall try to shrink the middle segment A[i:j] into the single element A[f] .

We form a loop, with i=f=j as the exit test, and vacuously initialized the invariant by

```
(i,j) ← (1,n)
```

Since $1 \leq f \leq n$, the range strategy suggests the invariant $1 \leq i \leq f \leq j \leq n$ and increasing i and/or decreasing j , ensuring termination of the loop. (Since $i \leq f < j$, the exit test may be simplified to $i=j$.)

3.2
```
begin assert 1≤f≤n
      (i,j) ← (1,n)
      loop assert A[1:i-1]≤A[i:j]≤A[j+1:n] , 1≤i≤f≤j≤n
           until i=j
           assert A[1:i-1]≤A[i:j]≤A[j+1:n] , 1≤i≤f≤j≤n , i≠j
              achieve A'[1:i'-1]≤A'[i':j']≤A'[j'+1:n] , i<i'≤f,
                      j'=j
           or achieve A'[1:i'-1]≤A'[i':j']≤A'[j'+1:n] , f≤j'<j,
                      i'=i
           repeat
      end
```

In the disjunctive achieve we have omitted the possibility that both i and j are updated. Since $i<i'$ and $j'<j$ and we have already asserted the desired relation for $A[1:i-1]$ and $A[j+1:n]$, we can separate out those parts of the achieve. Thus, after also substituting for i' and j' , we obtain

```
     achieve A'[1:i-1]          ≤A'[i':j]≤A'[j+1:n] ,
                     A'[i:i'-1]≤A'[i':j] ,
             i<i'≤f , j'=j
  or achieve A'[1:i-1]≤A'[i:j']≤          A'[j+1:n] ,
                     A'[i:j']≤A'[j'+1:j] ,
             f≤j'<j , i'=i .
```

We would like to apply the if-then-else-fi strategy to this disjunctive goal, but we want to first unify the two disjuncts as much as possible. We notice that by introducing a new variable g , where $g=i'-1$ in the first disjunct and $g=j'$ in the second, the second lines of each disjunct are unified to $A'[i:g] \leq A'[g+1:j]$. The goal, then, is transformed into

```
assert A[1:i-1]≤A[i:j]≤A[j+1:n] , 1≤i≤f≤j≤n , i≠j
   achieve A'[1:i-1]        ≤A'[g+1:j]≤A'[j+1:n] ,
                    A'[i:g]≤A'[g+1:j] ,
           i<g+1≤f , i'=g+1 , j'=j
or achieve A'[1:i-1]≤A'[i:g]≤          A'[j+1:n] ,
                    A'[i:g]≤A'[g+1:j] ,
           f≤g<j , j'=g , i'=i  .
```

We can extract from both disjuncts the identical conjunct $A'[i:g] \leq A'[g+1:j]$, and since we have asserted $i \leq f \leq j$ and want $i \leq g < f$ in the first disjunct, and $f \leq g < j$ in the second, we can also extract $i \leq g < j$. In the first disjunct we wish to maintain $A'[1:i-1] \leq A'[g+1:j] \leq A'[j+1:n]$ and in the second $A'[1:i-1] \leq A'[i:g] \leq A'[j+1:n]$. It is therefore sufficient to require the preservation of both, that is (since $i \leq g < j$), $A'[1:i-1] \leq A'[i:j] \leq A'[j+1:n]$. So

achieve $A'[1:i-1] \leq A'[i:j] \leq A'[j+1:n]$,
$A'[i:g] \leq A'[g+1:j]$, $i \leq g < j$,
$(i'=g+1 \leq f \wedge j'=j) \vee (i'=i \wedge j'=g \geq f)$.

To preserve the first conjunct we shall require that the multi-sets $A'[1:i-1]$, $A'[i:j]$ and $A'[j+1:n]$ equal $A[1:i-1]$, $A[i:j]$ and $A[j+1:n]$, respectively. We have

achieve $A'[1:i-1]=A[1:i-1]$, $A'[i:j]=A[i:j]$, $A'[j+1:n]=A[j+1:n]$,
$A'[i:g] \leq A'[g+1:j]$, $i \leq g < j$,
$(i'=g+1 \leq f \wedge j'=j) \vee (i'=i \wedge j'=g \geq f)$.

Since we shall only exchange within the range $[i:j]$, these equalities will be maintained and we omit them in the sequel. We now split our goal into two consecutive subgoals

achieve $A'[i:g] \leq A'[g+1:j]$, $i \leq g < j$
achieve $A'=A$, $g'=g$, $i'=g+1$, $j'=j$, $g+1 \leq f$
or achieve $A'=A$, $g'=g$, $i'=i$, $j'=g$, $g \geq f$.

Since $g+1 \leq f$ and $g \geq f$ are mutually exclusive, we apply the if-then-else-fi strategy to the second subgoal obtaining

3.3
assert $A[1:i-1] \leq A[i:j] \leq A[j+1:n]$, $1 \leq i \leq f \leq j \leq n$, $i \neq j$
achieve $A'[i:g] \leq A'[g+1:j]$, $i \leq g$, $g+1 \leq j$
if $g \geq f$ then $j \leftarrow g$ else $i \leftarrow g+1$ fi

So far we have succeeded in synthesizing the shrinking of the middle segment $A[i:j]$ by first dividing it around some point g and then expanding either the left segment $A[1:i-1]$ to $A[1:g]$ ($i \leftarrow g+1$ when $g+1 \leq f$) or the right segment $A[j+1:n]$ to $A[g+1:n]$ ($j \leftarrow g$ when $g \geq f$) .

The remaining subgoal is actually a specification of Hoare's PARTITION subroutine. Its purpose is to rearrange an array segment $A[i:j]$ so that there exists some point g which partitions $[i:j]$

into two ordered parts.

We strengthen the current goal, by introducing a new program variable h :

achieve $A'[i:g] \leqslant A'[h+1:j]$, $i \leqslant g$, $h+1 \leqslant j$, $g=h$.

(Since the conjunct $g+1 \leqslant j$ was closely connected with the range $[g+1:j]$, both were transformed.) It is possible, for example, to let $g=i$ and search for the minimum of $A[i:j]$. However in order to derive a more efficient program, we use the (tailor-made)

fact $u \leqslant y \leqslant v \supset u \leqslant v$

which allows us to replace the nested quantified expression $A'[i:g] \leqslant A'[h+1:j]$ with a conjunction of two singly quantified expressions $A'[i:g] \leqslant y$ and $y \leqslant A'[h+1:j]$. We obtain the stronger goal

achieve $A'[i:g] \leqslant y \leqslant A'[h+1:j]$, $i \leqslant g$, $h+1 \leqslant j$, $g=h$.

We now apply the loop-until-repeat strategy, testing for $g=h$, and must initialize

achieve $A'[i:g] \leqslant y \leqslant A'[h+1:j]$, $i \leqslant g$, $h+1 \leqslant j$.

Taking $i=g$ and $h+1=j$, in order to eliminate the quantifiers, we are left with

achieve $A'[i] \leqslant y$, $y \leqslant A'[j]$, $i=g$, $h+1=j$

or, by strengthening the first conjunct,

achieve $y=A'[i]$, $A'[i] \leqslant A'[j]$, $g=i$, $h=j-1$.

A simple application of the if-then-else-fi strategy gives the initialization

```
if A[i]≤A[j] then null else exchange(A[i], A[j]) fi
(y,g,h) ← (A[i], i, j-1)
```

It is now possible to determine bounds for g and h and use the range strategy, since $g^0=i \leqslant g^f=h^f \leqslant j-1=h^0$. We assume $i \leqslant g \leqslant g^f = h^f \leqslant h \leqslant j-1$, and therefore keep $i \leqslant g \leqslant h \leqslant j-1$ invariant. Furthermore, we require that either g increases or h decreases, and the loop must terminate.

We have, so far, the outer loop body:

3.4
```
assert A[1:i-1]≤A[i:j]≤A[j+1:n] , 1≤i≤f≤j≤n , i≠j
if A[i]≤A[j] then null else exchange(A[i], A[j]) fi
(y,g,h) ← (A[i], i, j-1)
assert y=A[i]≤A[j] , (g,h)=(i,j-1)
loop assert A[i:g]≤y≤A[h+1:j], i≤g≤h≤j-1
     until g=h
     achieve A'[i:g']≤y'≤A'[h'+1:j] , g≤g'≤h'≤h , (g<g' ∨ h'<h)
     repeat
if g≥f then j ← g else i ← g+1 fi
```

For the inner loop body, we have the subgoal:

```
assert A[i:g]≤y≤A[h+1:j] , i≤g<h≤j-1
achieve A'[i:g']≤y'≤A'[h'+1:j] , g≤g'≤h'≤h , (g<g' ∨ h'<h) .
```

Protecting what has been asserted gives

```
achieve A'[i:g]=A[i:g] , y'=y , A'[h+1:j]=A[h+1:j] ,
        A'[g+1:g']≤y≤A'[h'+1:h] ,
        g≤g'≤h'≤h , (g<g' ∨ h'<h) .
```

Now eliminating quantifiers (and omitting the first line) leaves

```
   achieve A'[g+1]≤y  , g'=g+1 , h'=h
or achieve  y≤A'[h] , h'=h-1 , g'=g
or achieve A'[g+1]≤y≤A'[h]  , g'≤h' , g'=g+1 , h'=h-1 .
```

The if-then-else-fi strategy reduces this to

```
assert A[i:g]≤y≤A[h+1:j] , i≤g<h≤j-1
     if A[g+1]≤y then g ← g+1
else if y≤A[h] then h ← h-1
else assert A[h]<y<A[g+1]
        achieve A'[g+1]≤y , g'=g+1 , h'=h
     or achieve  y≤A'[h] , h'=h-1 , g'=g
     or achieve A'[g+1]≤y≤A'[h]  , g+1≤h-1 , g'=g+1 , h'=h-1
     fi fi
```

The last disjunct in this subgoal is prefered since it advances more towards the exit test g=h . Written in full, we have

```
assert A[i:g]≤y≤A[h+1:j] , i≤g<h≤j-1 , A[h]<y<A[g+1]
achieve A'[i:g]=A[i:g] , y'=y , A'[h+1:j]=A[h+1:j] ,
        A'[g+1]≤y≤A'[h] , g+1≤h-1 , g'=g+1 , h'=h-1 .
```

The conjunct g+1≤h-1 can be proven true, since A[h]<A[g+1] implies h≠g+1 and we also know g<h . It is sufficient to

achieve A'[i:g]=A[i:g] , y'=y , A'[h+1:j]=A[h+1:j] ,
A'[g+1]=A[h] , A'[h]=A[g+1] , g'=g+1 , h'=h-1 .

Since neither g+1 nor h are in the ranges [i:g] and [h+1:j] , this is achievable by

```
exchange(A[h],A[g+1])
(g,h) ← (g+1 , h-1)
```

Note that both g+1 and h are within the range [i:j] and the unwritten conjuncts A'[1:i-1]=A[1:i-1] , A'[i:j]=A[i:j] , A'[j+1:n]=A[j+1:n] are also satisfied.

The complete program is:

3.5

```
begin constant integer f,n
      assert 1≤f≤n
      variable real array A[1:n]
      variable integer i,j
      (i,j) ← (1,n)
      loop assert A[1:i-1]≤A[i:j]≤A[j+1:n] , 1≤i≤f≤j≤n
           until i=j
           purposel A'[1:i'-1]≤A'[i':j']≤A'[j'+1:n] ,
                    i≤i'≤f≤j'≤j , (i<i' ∨ j'<j)
           variable integer g,h
           variable real y
           purpose2 A'[1:i-1]=A[1:i-1] , A'[i:j]=A[i:j] ,
                   A'[j+1:n]=A[j+1:n] , A'[i:g]≤A'[g+1:j] , i≤g<j
           if A[i]≤A[j] then null else exchange(A[i], A[j]) fi
           (y,g,h) ← (A[i], i, j-1)
           loop assert A'[1:i-1]=A[1:i-1] , A'[i:j]=A[i:j] ,
                                A'[j+1:n]=A[j+1:n] ,
                       A'[i:g]≤y≤A'[h+1:j] , i≤g≤h<j
                until g=h
                purpose3 A'[i:g']≤y'≤A'[h'+1:j] , g≤g'≤h'≤h ,
                         (g<g' ∨ h'<h)
                     if A[g+1]≤y then g ← g+1
                else if y≤A[h] then h ← h-1
                else exchange(A[h], A[g+1])
                     (g,h) ← (g+1, h-1)
                     fi fi eop3
                repeat
```

```
            assert A'[1:i-1]=A[1:i-1] , A'[i:j]=A[i:j] ,
                                  A'[j+1:n]=A[j+1:n] ,
                        A'[i:g]≤y≤A'[g+1:j] , i≤g = h<j eop2
            purpose4 A'=A , g'=g , (i'=g+1≤f∧j'=j)∨(i'=i∧j'=g≥f)
            if g≥f then j ← g else i ← g+1 fi eop4 eop1
            repeat
        assert A'[1:f-1]≤A'[f]≤A'[f+1:n] , A'[1:n]=A[1:n]
        end
```

IV. ABSTRACTION

In this section we illustrate the possibility of abstracting completed programs, and later applying the subroutines and techniques obtained.

For example, after successfully synthesizing a binary search in a monotonic array (possibly interactively, see Floyd [1971]), we would like to somehow extract the underlying search technique. We have, say, the annotated search program:

```
begin constant integer n; real x; real array X[1:n]
      assert (∃k,ℓ)(1≤k≤n)(x=X[k]) , (∀k,ℓ)(1≤k≤ℓ≤n)(X[k]≤X[ℓ])
      variable integer z,y,h
      (z,y) ← (1,n)
      loop assert x≥X[z] , x≤X[y] , 1≤z≤y≤n
           until z≥y
           h ← (z+y)÷2
           if x≤X[h] then y ← h else z ← h+1 fi
           repeat
      assert x≥X[z] , x≤X[z] , 1≤z≤n
      end
```

Analyzing the proof of this program shows that the following conditions - derivable from the given input assertion - are sufficient for proving correctness:

(a) initialization. $x \geq X[1]$, $x \leq X[n]$, $1 \leq n$

(b) then path. none

(c) else path. $(\forall u)(\sim(x \leq X[u]) \supset x \geq X[u+1])$.

This condition is a "coverup" for the weaker

$$(\forall z,y)\,(x \geq X[z] \wedge x \leq X[y] \wedge 1 \leq z < y \leq n \wedge \sim(x \leq X[(z+y)\div 2]) \supset x \geq X[(z+y)\div 2+1])\ .$$

Furthermore, the particular semantics of the predicates $x \geq X[z]$ and $x \leq X[z]$ are not needed, nor the fact that the lower bound of the

range of X is the constant 1. We accordingly abstract the program by replacing 1 with ℓ , $x \geqslant X[z]$ with $p(z)$ and $x < X[z]$ with $q(z)$. Thus, we have abstracted the schema

4.2
```
begin constant integer ℓ,n
      assert ℓ≤n , p(ℓ) , q(n) , (∀u)(~q(u) ⊃ p(u+1))
      variable integer z,y,h
      (z,y) ← (ℓ,n)
      loop assert p(z) , q(y) , ℓ≤z≤y≤n
           until z≥y
           h ← (z+y)÷2
           if q(h) then y ← h else z ← h+1 fi
           repeat
      assert p(z) , q(z) , ℓ≤z≤n
      end
```

This schema will solve a goal of the form

achieve $p(z)$, $q(z)$, $\ell \leqslant z$, $z \leqslant n$,

provided we can prove (or first achieve): $\ell \leqslant n$, $p(\ell)$, $q(n)$, and $(\forall u)(\sim q(u) \supset p(u+1))$.

Consider now our integer square-root problem. Our goal was

assert $0 \leqslant x$
achieve $z^2 \leqslant x$, $x < (z+1)^2$, $0 \leqslant z$.

(The conjunct $0 \leqslant z$ was suggested by the range strategy.) Matching this with the schema, we let $p(z)$ be $z^2 \leqslant x$, $q(z)$ be $x < (z+1)^2$ and ℓ be 0 .

The last conjunct achieved by the schema, $z \leqslant n$, has no counterpart. To apply the schema, we must have: $0 \leqslant n$, $0^2 \leqslant x$, $x < (n+1)^2$ and $(\forall u)(x \geqslant (u+1)^2 \supset (u+1)^2 \leqslant x)$. The second and fourth conditions are trivially proved, leaving the subgoal

achieve $0 \leqslant n$, $x < (n+1)^2$

to be achieved (e.g., by $n \leftarrow x$) before applying the schema. In this manner, we have arrived at an (unoptimized) binary search program for the integer square-root.

As a second example consider the search for an array maximum that we synthesized in Example 2. The solution may be straightforwardly abstracted to obtain the schema

5.1
```
begin assert ℓ≤n , p(ℓ,ℓ) , p(z,[ℓ:y1])∧~p(z,y1+1) ⊃
                                  p(y1+1,[ℓ:y1+1])
      (z,y1) ← (ℓ,ℓ)
      loop assert p(z,[ℓ:y1]) , ℓ≤z≤n , ℓ≤y1≤n
           until y1=n
           if p(z,y1+1) then null else z ← y1+1 fi
           y1 ← y1+1
           repeat
      assert p(z,[ℓ:n]) , ℓ≤z≤n , y1=n
      end
```

Here $p(z,[\ell:n])$ is short for $(\forall k)(\ell \leq k \leq n)(p(z,k))$. This schema can then be used for either maximum or minimum, both of which satisfy the input assertion for p .

If we are presented with the problem

achieve $z_2=X[z_1]$, $X[z_1] \leq X[i:j]$, $i \leq z_1 \leq j$

we could solve it by using the above schema and adding the conjunct $z_2=X[z_1]$ as an invariant throughout the loop (cf. the suggestion for optimization at the end of the example) or simply by assigning $z_2 \leftarrow X[z_1]$ after achieving $X[z_1] \leq X[i:j]$. In a sort program, if we have a subgoal

achieve $X'[i] \geq X'[i:n]$, $X'[i:n]=X[i:n]$

we might separate it into a search for a maximum followed by an exchange:

achieve $X[z] \geq X[i:n]$, $i \leq z \leq n$
achieve $X'[i]=X[z]$, $X'[i:n]=X[i:n]$.

Another possibility for abstraction: first synthesize an iterative version of a recursive program, say factorial, and then abstract it to a more general recursion-to-iteration translation schema such as those of Darlington and Burstall [1973].

V. CONCLUSION

Clearly the work described in this paper is but a small step toward the distant, albeit important, goal of fully or semi-automated programming. The aspects we have dealt with and other facets of the overall goal are the subject of much current research.

Some of the still outstanding problems are typical of artificial intelligence research. Specific items in need of further investigation are: providing the system with a "sense of direction", intro-

ducing more complex data structures, developing tools for the evaluation and comparison of program efficiency, formulating more general termination strategies; determining which program segments are worthy of remembering and formalizing the details of their abstraction, formulating precise criteria for schema application, and specifying methods for adapting a schema which does not quite fit.

A programming system such as that suggested here is obviously limited by the power of the theorem prover used. We are planning the implementation of the strategies described in QLISP (Reboh and Sacerdoti [1973]) using Waldinger and Levitt's [1974] program verification system.

ACKNOWLEDGEMENT

We are deeply indebted to Richard Waldinger for stimulating discussions and to Shmuel Katz and Adi Shamir for their critical reading of the manuscript.

REFERENCES

Buchanan, J.R. and D.C. Luckham [Mar. 1974], On automating the construction of programs, Memo AIM-236, Stanford A.I. Lab.

Conway, R. and D. Gries [1973], An Introduction to Programming: A Structured Approach, Winthrop.

Dahl, O.S., E.W. Dijkstra and C.A.R. Hoare [1972], Structured Programming, Academic Press.

Darlington, J. and R.M. Burstall [Aug. 1973], A system which automatically improves programs, Adv. Papers 3d. Intl. Conf. on Artificial Intelligence, Stanford Univ.

Deutsch, L.P. [May 1973], An interactive program verifier, Memo CSL-73-1, Xerox, Palo Alto Research Center.

Dijkstra, E.W. [1968], A constructive approach to the problem of program correctness, BIT, Vol. 8, No. 3, 174-186.

Dijkstra, E.W. [Aug. 1971], A short introduction to the art of programming, Report EWD316, Technological Univ., Eindhoven.

Dijkstra, E.W. [1973], unpublished lectures, Intl. Summer School on Structured Programming and Programmed Structures, Munich.

Dijkstra, E.W. [April 1975], Guarded commands, non-determinancy and a calculus for the derivation of programs, Proc. of Intl. Conf. on Reliable Software, Los Angeles.

Duran, J.W. [Sept. 1974], Loop invariants and automatic program synthesis, Report SESLTR-6, Software Engineering and Systems Lab., Univ. of Texas, Austin.

Floyd, R.W. [1967], Assigning meanings to programs, Proc. of a Symposium in Applied Mathematics, Vol. 19, (J.T. Schwartz, ed.), AMS, 19-32.

Floyd, R.W. [Aug. 1971], Toward interactive design of correct programs, Proc. of IFIP Congress 1971, Ljubljana, North Holland.

Gerhart, S.L. [Apr. 1975], Knowledge about programs: A model and a case study, Proc. of Intl. Conf. on Reliable Software, Los Angeles.

Green, C.C., R.J. Waldinger, D.R. Barstow, R. Elschlager, D.B. Lenat, B.P. McCune, D.E. Shaw, and L.I. Steinberg [Aug. 1974], Progress report on program-understanding systems, Memo AIM-240, Stanford A.I. Lab.

Gries, D. [Nov. 1974], On structured programming - a reply to Semoliar, CACM, Vol. 17, No. 11, 655-657.

Hoare, C.A.R. [July 1961], Algorithm 63 (Partition) and Algorithm 65 (Find), CACM, Vol. 4, No. 7, 321-322.

Hoare, C.A.R. [Oct. 1969], An axiomatic basis of computer programming, CACM, Vol. 12, No. 10, 576-580, 583.

Hoare, C.A.R. [Jan. 1971], Proof of a program: Find, CACM, Vol. 14, No. 1, 39-45.

Katz, S. and Z. Manna [Aug. 1973], A heuristic approach to program verification, Adv. Papers 3d. Intl. Conf. on Artificial Intelligence, Stanford Univ.

Manna, Z. [1974], Mathematical Theory of Computation, McGraw-Hill.

Manna, Z. and R.J. Waldinger [1975], Knowledge and reasoning in program synthesis, to appear in J. of Artificial Intelligence.

Reboh, R. and E. Sacerdoti [Aug. 1973], A preliminary QLISP manual, Tech. Note 81, A.I. Center, SRI, Menlo Park.

Sussman, G.J. [Aug. 1973], A computational model of skill acquisition, Ph.D. Thesis, Report AI-TR-297, A.I. Lab., MIT, Cambridge.

Waldinger, R.J. and K.N. Levitt [1974], Reasoning about programs, J. of Artificial Intelligence, Vol. 5, 235-316.

Wegbreit, B. [Feb. 1974], The synthesis of loop predicates, CACM, Vol. 17, No. 2, 102-112.

Wirth, N. [Apr. 1971], Program development by stepwise refinement, CACM, Vol. 14, No. 4, 221-227.

Wirth, N. [1973], Systematic Programming: An Introduction, Prentice Hall.