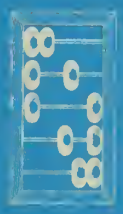
Report No. UIUCDCS-R-81-1011 UILU-ENG 81 1717

# THE EVOLUTION OF PROGRAMS: PROGRAM ABSTRACTION AND INSTANTIATION

by

Nachum Dershowitz

June 1981 NSF-MCS-79-04897



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

# THE EVOLUTION OF PROGRAMS: PROGRAM ABSTRACTION AND INSTANTIATION*

Nachum Dershowitz
Department of Computer Science
University of Illinois
Urbana, IL 61801

June 1981

*Research supported in part by the National Science Foundation under grant MCS-79-04897.

## ABSTRACT

Our goal is to develop techniques for abstracting a given set of programs into a program schema and for instantiating a given schema to satisfy concrete specifications. Abstraction and instantiation are two important phases in software development that allow programmers to apply knowledge learned in the solution of past problems when faced with a new situation. For example, from two programs using the binary-search method, one to compute quotients and another to compute cube-roots, an abstract schema can be derived that embodies the shared method and that can be instantiated to solve similar new problems.

We suggest the formulation of analogies as a basic tool in program abstraction. An analogy is first sought between the specifications of the given programs; this yields an abstract specification that may be instantiated to any of the given concrete specifications. The analogy is then used as a basis for transforming the existing programs into an abstract schema that represents the embedded technique, with the invariant assertions and correctness proofs of the given programs helping to verify and complete the analogy. A given concrete specification of a new problem may then be compared with the abstract specifications of the schema to suggest an instantiation of the schema that yields a correct program. These methods appear to be amenable to implementation.

A collection of program schemata is included in an appendix.



Keywords: abstraction, instantiation, program schemata, analogy, transformations, program manipulation, program development, program correctness

CR Categories: 3.62, 4.43, 5.24

# I. INTRODUCTION

> Chaque vérité que je trouvais étant une règle qui me servait après à en trouver d´autres, non seulement je vins à bout plusieurs que j´avais jugées autrefois très difficiles, mais il me sembla aussi, vers la fin, que je pouvais déterminer, en celles même que j´ignorais, par quels moyens, et jusques où, il était possible de les résoudre.
>
> - René Descartes, Discours de la Méthode

When confronted with a task, one often perceives some measure of resemblance between the given task and a previously accomplished one. In such a case, rather than "reinvent the wheel," one is likely to conserve effort by adapting the known solution of the old problem to the problem now at hand. After having solved several similar problems, one might come to formulate a general paradigm for solving such problems by highlighting the shared aspects of the individual instances and supressing their inconsequential or idiosyncratic particulars. The process of formulating a general scheme from concrete instances is termed abstraction; that of applying an abstract scheme to a particular problem is termed instantiation.

Abstraction and instantiation are two phases in the evolutionary cycle of many typical programs, a cycle that, in addition, often includes the debugging of early versions, modifications to meet amended specifications, and extensions for expanded capabilities. The more experience a programmer has had, the more programming methods he is likely to have assimilated, and the more judiciously he can apply them to new problems. After having written several similar programs, a programmer is apt to formulate for himself — and

perhaps for others as well — an abstract notion of the underlying principle and reuse it in solving related problems. Program schemata are a convenient form for remembering such programming knowledge. A schema may embody basic programming techniques or specialized strategies for solving some class of problems; its input-output specifications are stated in terms of abstract predicate, function, and constant symbols.

Our goal is to formalize this aspect of programming by developing automatable techniques for abstracting a given set of concrete programs into a program schema and for instantiating a schema to satisfy a given concrete specification. The programs are assumed to be annotated with an output specification (stating the desired relationship between the input and output variables upon termination of the program), an input specification (defining the set of legal inputs on which the program is intended to operate), and invariant assertions (relations that are known to always hold at specific points in the program for the current values of variables) demonstrating its correctness.

To date there has been a limited amount of research on program abstraction. The STRIPS system (Fikes, Hart, and Nilsson [1972]) generalized the loop-free robot plans that it generated; the HACKER system (Sussman [1975]) "subroutinized" and generalized the "blocks-world" plans it synthesized, executing the plan to determine which program constants could be abstracted. Dershowitz and Manna [1975] suggest using the proof of correctness of a program to guide the abstraction process.

A number of researchers have dealt with the use of program schemata. Dijkstra [1972] maintains that theorems about schemata are unconsciously invoked by the programmer. Wirth [1975] illustrates the use of basic schemata in the systematic development of programs. Gerhart [1975], Gerhart and Yelowitz [1976], and Yelowitz and Duncan [1977] have all advocated and illustrated the use of schemata as a powerful programming tool. Other examples of the use of abstract algorithms as first steps in the development of programs are Darlington [1978], Deussen [1979], Duncan and Yelowitz [1979], and Lee, deRoever, and Gerhart [1979]. In a similar vein, Plaisted [1980] demonstrates how abstractions may be used, with great effectivity, for theorem proving. An approach to the specification and verification of program schemata is given in Misra [1978].

We suggest the formulation of analogies as a basic tool in program abstraction. First, an analogy is sought between the output specifications of the given programs. This yields an abstract output specification that may be instantiated to any of the given concrete specifications. The analogy is then used as a basis for transforming the existing programs into an abstract schema that represents the embedded technique. The invariant assertions and correctness proofs of the given programs are used to extend and complete the analogy.

A schema, derived in this manner, is usually not applicable to all possible instantiations of its specifications. In that case, the schema is accompanied by an input specification containing conditions that must be satisfied by the instantiation in order to guarantee correctness. These

preconditions may be derived from the verification conditions or correctness proof of the schema. The abstract specifications of the schema may then be compared with a given concrete specification of a new problem. By formulating an analogy between the two specifications, an instantiation is found that yields a concrete program when applied to the schema. If the instantiation satisfies the preconditions, then the correctness of the new program is guaranteed. If not, analysis of the unsatisfied conditions may suggest modifications that will lead to a correct program.

The importance of analogical reasoning has been stressed by many, from Descartes to Polya. For a review of psychological theories of analogical reasoning see Sternberg [1977]. The use of analogy in automated problem solving in general, and theorem proving in particular, was proposed by Kling [1971]. Other works employing analogy as an implement in problem solving include Brown [1976], Chen and Findler [1976], McDermott [1979], and Winston [1980]. Using analogies to guide the modification of programs has been investigated by Manna and Waldinger [1975], Dershowitz and Manna [1977], Ulrich and Moll [1977], and Dershowitz [1978].

In the next section we present several illustrations of our approach. Some schemata that have appeared in the literature are collected in the Appendix.

## II. EXAMPLES

This section contains three detailed examples of program abstraction and instantiation. Each example begins with two concrete programs, annotated with their specifications and invariant assertions. Then, the programs are abstracted to obtain a more general program schema. Finally, each schema is instantiated to satisfy a third concrete specification.

### Example 1: Extremum Schema

Consider the following program:

```
P1: begin
      comment  minimum value program
      assert nεℕ
      (z,i) := (A[0],0)
      loop  assert z≤A[0:i], iεℕ
            until i=n
            i := i+1
            if A[i]<z then z := A[i] fi
            repeat
      assert z≤A[0:n]
      end,
```

where $\mathbb{N}$ is the set of nonnegative integers and for predicate $\alpha$ the construct $\alpha[u:w]$ is shorthand for $(\forall v \in \mathbb{Z})(u \leq v \leq w)\alpha(v)$, i.e. $\alpha$ holds for all integers between u and w. The output specification of this program is given in the statement

**assert** $z \prec A[0:n]$;

it states that upon termination of program $P_1$ the value of z is less than or equal to any value appearing in the array segment A[0:n]. The input specification of this program is given in the initial assertion

**assert** $n \varepsilon \mathbb{N}$;

it states that the value of the input variable n must be a nonnegative integer. The statement

**assert** $z \prec A[0:i]$, $i \varepsilon \mathbb{N}$

at the head of the loop contains two relations — invariant assertions — that hold for the current values of the variables i and z each time control reaches that point in the program. The invariants are first made true by the multiple assignment

$(z,i) := (A[0],0)$.

The two loop-body statements

```
i := i+1
if A[i]<z then z := A[i] fi
```

maintain the truth of the invariants, i.e. assuming that the invariants are true before the statements are executed, the invariants remain true afterwards. The loop body is repeated (zero or more times) until the test

**until** $i=n$

becomes true; at that point the loop is left, and the output specification holds.

The second program is

```
Q1: begin
      comment  maximum position program
      assert k,ℓ∈Z, k≤ℓ
      (p,j) := (ℓ,ℓ)
      loop  assert B[p]≥B[j:ℓ], p,j∈Z
            while j>k
            j := j-1
            if ¬(B[p]≥B[j]) then p := j fi
            repeat
      assert B[p]≥B[k:ℓ]
      end,
```

where **Z** is the set of all integers. This program finds the position p of a maximal element in the array segment B[k:ℓ]; its output specification is

**assert** B[p]≥B[k:ℓ].

Its input specification

**assert** k,ℓ∈**Z**, k≤ℓ

requires that the two input variables k and ℓ be integers and that k not be greater than ℓ so that the array segment B[k:ℓ] is nonempty.

Both programs perform a linear search for an extremum in an array segment. Our task is to extract an abstract version of these two programs that captures the essence of the technique used, but that is not specific to either problem. The resultant schema can then be used as a model of linear search for the solution of future problems.

The first step in abstracting these two programs is to find an analogy between their respective output specifications, z≤A[0:n] and B[p]≥B[k:ℓ]. The obvious analogy is that where the specification of $P_1$ has 0, n, ≤, A, and z, the specification of $Q_1$ has k, ℓ, ≥, B, and B[p], respectively. This analogy is denoted by

$$0 \longleftrightarrow k$$
$$n \longleftrightarrow \ell$$
$$< \longleftrightarrow >$$
$$A \longleftrightarrow B$$
$$z \longleftrightarrow B[p].$$

Now abstract entities may be substituted for analogous parts. Each pair in the analogy is replaced by an abstract variable of the same kind: the scalar constant 0 in $P_1$ and the corresponding scalar input variable k in $Q_1$ may be replaced by an abstract input variable $\kappa$; the corresponding input variables n and $\ell$ may be replaced by $\lambda$; the predicate constants $<$ and $>$ are abstracted to a predicate variable $\alpha$; the input arrays A and B generalize to $\Delta$; and the variable z and variable expression B[p] generalize to a program variable $\mu$. (We shall use Greek letters to distinguish all abstract entities.)

Applying the transformations

$$0 \rightarrow \kappa$$
$$n \rightarrow \lambda$$
$$< \rightarrow \alpha$$
$$A \rightarrow \Delta$$
$$z \rightarrow \mu$$

to the output specification of $P_1$ yields the abstract specification

$$\textbf{assert } \alpha(\mu,\Delta[\kappa:\lambda]).$$

Such a set of transformations is called an abstraction mapping. The abstraction mapping

$$a \rightarrow \kappa$$
$$\ell \rightarrow \lambda$$
$$\geq \rightarrow \alpha$$
$$B \rightarrow \Delta$$
$$p \rightarrow pos(\Delta,\mu),$$

where $pos(\Delta,\mu)$ is a function that returns the position of some occurrence of $\mu$ in the array $\Delta$, yields

$$\textbf{assert } \alpha(\Delta[pos(\Delta,\mu)],\Delta[\kappa:\lambda])$$

when applied to the output specification of $Q_1$. This simplifies to the same abstract specification

$$\textbf{assert } \alpha(\mu,\Delta[\kappa:\lambda]),$$

since by definition $\Delta[pos(\Delta,\mu)]=\mu$.

In general there are several possible ways in which an expression of the form f(q,r) can be compared with an expression h. If h is of the form g(s,t), then the imitating mapping $f\rightarrow g$, $q\rightarrow s$, and $r\rightarrow t$ suggests itself. If f has an inverse $f^{-}$ in its first argument, i.e. $f(f^{-}(u,v),v)=u$, then the inverting mapping $q\rightarrow f^{-}(h,r)$ would work. If f has an identity element $f^{0}$ in its first argument, i.e. $f(f^{0},v)=v$, then the collapsing mapping $q\rightarrow f^{0}$ and $r\rightarrow h$ is possible. Another possibility is the projecting mapping $f\rightarrow\pi$ and $r\rightarrow h$, where $\pi$ projects its first argument, i.e. $\pi(u,v)=u$. (Similar mappings work for other than the first argument.) For a discussion of a second-order pattern-matcher that uses imitating and projecting mappings and its application to program manipulation see Huet and Lang [1978].

In our example, to compare $z\leq A[0:n]$ with $B[p]\geq B[k:\ell]$, the imitating mapping $0\leftrightarrow a$, $n\leftrightarrow b$, and $z\leq A[u]\leftrightarrow B[p]\geq B[u]$ is first used (eliminating the quantifiers [0:n] and [k:ℓ]). To compare $z\leq A[u]$ with $B[p]\geq B[u]$, another imitating mapping, viz. $\leq\leftrightarrow\geq$, $z\leftrightarrow B[p]$, and $A[u]\leftrightarrow B[u]$, is used. An inverting mapping is needed, for example, to abstract B[p] into $\mu$ for $Q_1$: since B is abstracted into $\Delta$, we need to map $\Delta[p]$ into $\mu$; since pos is an inverse of the

array access function, we get $p \rightarrow pos(\Delta,\mu)$. An example of a projecting mapping would be $\Delta \rightarrow \pi$ and $p \rightarrow \mu$.

The next step in abstracting $P_1$ and $Q_1$ is to apply the mappings to the corresponding programs. Applying the first set of transformations to $P_1$ yields

```
S1: begin
      comment  tentative extremum schema
      suggest λεℕ
      (μ,i) := (Δ[κ],κ)
      loop  suggest α(μ,Δ[κ:i]), iεℕ
            until i=λ
            i := i+1
            if Δ[i]<μ then μ := Δ[i] fi
            repeat
      suggest α(μ,Δ[κ:λ])
      end.
```

Had all the transformations been of variables, as are $n \rightarrow \lambda$, $A \rightarrow \Delta$, and $z \rightarrow \mu$, then applying the transformations to all occurrences of those variables in the annotated program would of necessity have resulted in a correct schema. But since the abstraction mapping involves the transformation of constant symbols as well, viz. 0 and $<$, this schema is not necessarily correct. We have therefore replaced the assertions with "suggestions" containing those relations that we would like to be invariants.

In general, a global transformation, where an input variable (e.g. n or A) is systematically replaced — in the assertions as well as in the code — by some function of only input variables, will always yield a program (or program schema) that satisfies the transformed input-output specifications. Similarly, systematically replacing a program (or output) variable (e.g. z) by a function of (input or program) variables preserves correctness with respect to the specifications. However, global transformations of constants (e.g. 0 and $<$) might be overzealous or incomplete, and they are not guaranteed to result in a program satisfying the specifications. That is because the

correctness of the original program may have depended on properties of these constants that do not hold for their transformed counterparts. There could, for example, be unrelated occurrences of 0 in $P_1$, say for the purpose of illustration that the exit test were $i+0=\lambda$ or that there was an additional loop-body assignment $i:=i+0$, in which cases the transformation $0\rightarrow\kappa$ would be inappropriate. Or the predicate symbol $\leqslant$ might not appear explicitly in the program code at all, as indeed it does not, in which case the transformation $\leqslant\rightarrow\alpha$ would be insufficient.

One must therefore examine the schema's verification conditions to determine under what assumptions the suggestions may in fact be invariants. First, consider the loop-exit path

**suggest** $\alpha(\mu,\Delta[\kappa:i])$, $i\varepsilon\mathbb{N}$
**assert** $i=\lambda$
**suggest** $\alpha(\mu,\Delta[\kappa:\lambda])$.

(The first suggestion is the transformed loop invariant that we are assuming was true when control was last at the head of the loop; the second statement asserts that the exit test was true at that time and therefore the loop was exited; the final suggestion contains the desired output relations with respect to which we are trying to prove this schema correct.) For this path to be correct, the loop invariants in the first suggestion, together with the exit test of the assertion, must imply that the desired output relation in the second suggestion holds, i.e. the verification condition for the exit path is

$$\alpha(\mu,\Delta[\kappa:i]) \wedge i\varepsilon\mathbb{N} \wedge i=\lambda \supset \alpha(\mu,\Delta[\kappa:\lambda]).$$

Indeed, if $\alpha(\mu,\Delta[\kappa:i])$ holds and $i=\lambda$, then $\alpha(\mu,\Delta[\kappa:\lambda])$ holds as well. Thus, if we can establish that $\alpha(\mu,\Delta[\kappa:i])$ and $i\varepsilon\mathbb{N}$ are loop invariants, then the schema is correct.

Next, we consider the initialization path

**suggest** $\lambda\varepsilon\mathbb{N}$
$(\mu,i) := (\Delta[\kappa],\kappa)$
**suggest** $\alpha(\mu,\Delta[\kappa{:}i])$, $i\varepsilon\mathbb{N}$.

Its verification condition is

$$\lambda\varepsilon\mathbb{N} \supset \alpha(\Delta[\kappa],\Delta[\kappa{:}\kappa]) \wedge \kappa\varepsilon\mathbb{N}.$$

(This condition is obtained from the path by substituting the new values $\Delta[\kappa]$ and $\kappa$ assigned to the variables $\mu$ and i, respectively, for occurrences of the variables in the suggestion following the assignment.) The subterm $\Delta[\kappa{:}\kappa]$ simplifies to $\Delta[\kappa]$. For cosmetic purposes, we shall replace the expression $\Delta[\kappa]$ in $\alpha(\Delta[\kappa],\Delta[\kappa])$ by the universally quantified u and just write $\alpha(u,u)$, i.e. $\alpha$ must be reflexive. Since there is no way of showing

$$\kappa\varepsilon\mathbb{N}$$

or

$$\alpha(u,u)$$

to hold, they are both left as preconditions for the schema to be correct.

The loop-body path verification condition is composed of two cases, one for each possible outcome of the conditional test. In the case when the test fails, the then-branch is skipped:

**suggest** $\alpha(\mu,\Delta[\kappa{:}i])$, $i\varepsilon\mathbb{N}$
**assert** $\neg(i{=}\lambda)$
i := i+1
**assert** $\neg(\Delta[i]{<}\mu)$
**suggest** $\alpha(\mu,\Delta[\kappa{:}i])$, $i\varepsilon\mathbb{N}$.

To verify this path we must show that the loop invariants continue to hold if the exit test is false, i is incremented, and the conditional test is false. The corresponding verification condition is

$$\alpha(\mu,\Delta[\kappa{:}i]) \wedge i\varepsilon\mathbb{N} \wedge \neg(i=\lambda) \wedge \neg(\Delta[i+1]<\mu) \supset \alpha(\mu,\Delta[\kappa{:}i+1]) \wedge i+1\varepsilon\mathbb{N}.$$

Clearly, if $i\varepsilon\mathbb{N}$ before executing $i:=i+1$, then $i\varepsilon\mathbb{N}$ after, i.e. $i\varepsilon\mathbb{N}$ implies $i+1\varepsilon\mathbb{N}$. We must also show $\alpha(\mu,\Delta[\kappa{:}i+1])$. But it is already assumed that $\alpha(\mu,\Delta[\kappa{:}i])$; all that remains to show is $\alpha(\mu,\Delta[i+1])$. The only possibly relevant assumption (relating $\mu$ and $\Delta[i+1]$) is $\neg(\Delta[i+1]<\mu)$, i.e. $\neg(\Delta[i+1]<\mu)$ should imply $\alpha(\mu,\Delta[i+1])$. Of course, since $\alpha$ is an abstract predicate, there is no reason for that to be the case. Accordingly, we look for an abstraction mapping that will unify the two relations:

$$\neg(\Delta[i+1]<\mu) \rightarrow \alpha(\mu,\Delta[i+1]).$$

Inverting gives

$$\Delta[i+1]<\mu \rightarrow \neg\alpha(\mu,\Delta[i+1]);$$

imitating leaves

$$u<v \rightarrow \neg\alpha(v,u).$$

Applying this additional transformation to (the loop body of) the schema, we get the conditional statement

$$\textbf{if } \neg\alpha(\mu,\Delta[i]) \textbf{ then } \mu := \Delta[i] \textbf{ fi}.$$

Now the verification condition for the case when the test is false carries through, and it remains to verify the other case when $\neg\alpha(\mu,\Delta[i])$ is true:

**suggest** $\alpha(\mu,\Delta[\kappa{:}i])$, $i\varepsilon\mathbb{N}$
**assert** $\neg(i=\lambda)$
$i := i+1$
**assert** $\neg\alpha(\mu,\Delta[i])$
$\mu := \Delta[i]$
**suggest** $\alpha(\mu,\Delta[\kappa{:}i])$, $i\varepsilon\mathbb{N}$.

The verification condition for this case is

$$\alpha(\mu,\Delta[\kappa{:}i]) \wedge i\varepsilon\mathbb{N} \wedge \neg(i=\lambda) \wedge \neg\alpha(\mu,\Delta[i+1]) \supset \alpha(\Delta[i+1],\Delta[\kappa{:}i+1]) \wedge i+1\varepsilon\mathbb{N}.$$

Again, the invariant $i\varepsilon\mathbb{N}$ is clearly maintained, and since we are already assuming that $\alpha$ is reflexive, i.e. $\alpha(\Delta[i+1],\Delta[i+1])$, we need only ascertain $\alpha(\Delta[i+1],\Delta[\kappa{:}i])$. Since there is no way to prove this to hold for all $\alpha$, it is left as a precondition:

$$\alpha(\mu,\Delta[\kappa{:}i]) \wedge i\varepsilon\mathbb{N} \wedge \neg(i=\lambda) \wedge \neg\alpha(\mu,\Delta[i+1]) \supset \alpha(\Delta[i+1],\Delta[\kappa{:}i])$$

For convenience, we shall replace this with (the more general condition)

$$\alpha(w,u) \wedge \neg\alpha(w,v) \supset \alpha(v,u).$$

Finally, we consider the verification condition for termination:

$$\lambda\varepsilon\mathbb{N} \supset (\exists u\varepsilon\mathbb{N})\kappa+u=\lambda$$

(i.e. the exit test $i=\lambda$ will eventually hold for some value $\kappa+u$ of $i$). Since $\kappa,\lambda\varepsilon\mathbb{N}$ is assumed to hold ($\lambda\varepsilon\mathbb{N}$ appears in the suggested input specification and $\kappa\varepsilon\mathbb{N}$ is a precondition), this termination condition is equivalent to

$$\kappa\leq\lambda.$$

This too is left as a precondition.

Applying the complete abstraction mapping

$$0 \rightarrow \kappa$$
$$n \rightarrow \lambda$$
$$\leq \rightarrow \alpha$$
$$A \rightarrow \Delta$$
$$z \rightarrow \mu$$
$$u<v \rightarrow \neg\alpha(v,u)$$

to program $P_1$, our final version of the schema is

$S_1$: **begin**
**comment** extremum schema
**assert** $\alpha(u,u)$, $\alpha(w,u) \wedge \neg\alpha(w,v) \supset \alpha(v,u)$, $\kappa,\lambda \in \mathbb{N}$, $\kappa \leq \lambda$
$(\mu,i) := (\Delta[\kappa],\kappa)$
**loop** **assert** $\alpha(\mu,\Delta[\kappa:i])$, $i \in \mathbb{N}$
**until** $i=\lambda$
$i := i+1$
**if** $\neg\alpha(\mu,\Delta[i])$ **then** $\mu := \Delta[i]$ **fi**
**repeat**
**assert** $\alpha(\mu,\Delta[\kappa:\lambda])$
**end.**

Now that we have verified the conditions for each of the paths in the program, the suggestions have been replaced by assertions. The preconditions are given in the input assertion; any instantiation that satisfies them is guaranteed to yield a correct program. Obviously, the predicate $\alpha$ that appears in the schema should be instantiated to a primitive predicate available in the target language; otherwise it must be replaced by something equivalent for the schema to yield an executable program. Likewise, the constants $\Delta$, $\kappa$, and $\lambda$ must be replaced with primitives.

Recall that the given output specification of program $P_1$ was

**assert** $z \prec A[0:n]$.

Actually, a more realistic specification for a search for a minimum would have included the relation $z \varepsilon A[0:n]$, i.e. not only should z be no larger than any element of A[0:n], but it should be one of those elements. If we apply the abstraction mapping that we have found to this additional specification as well, we get the abstract relation $\mu \varepsilon \Delta[\kappa:\lambda]$ which indeed may be shown to hold for the schema we have derived. We also note that had we applied the corresponding abstraction mapping to $Q_1$ instead of to $P_1$, a somewhat different, but equally valid, schema would have resulted.

The schema $S_1$ can be instantiated to find the position or value $\mu$ of either the minimum or maximum of any function $\Delta$ over some domain of nonnegative integers in the range $[\kappa:\lambda]$. For instance, say we want to find the position m of the minimum of some function f for the odd integers in the interval [1:100]. Comparing this goal

**achieve** $(\forall u \varepsilon [1:100])(\text{odd}(u) \supset f(m) \prec f(u))$

with the abstract output specification of the schema

**assert** $\alpha(\mu, \Delta[\kappa:\lambda])$,

suggests first applying the imitating mappings $\kappa \rightarrow 1$, $\lambda \rightarrow 100$, and $\alpha(\mu, \Delta[u]) \rightarrow \text{odd}(u) \supset f(m) \prec f(u)$. The last of these mappings can be accomplished by the projecting and imitating mappings $\Delta \rightarrow \pi$, $\alpha(v,u) \rightarrow \text{odd}(u) \supset f(v) \prec f(u)$, and $\mu \rightarrow m$, where $\pi$ is the identity function.

Applying the instantiation mapping

$$\kappa \rightarrow 1$$
$$\lambda \rightarrow 100$$
$$\Delta \rightarrow \pi$$
$$\alpha(v,u) \rightarrow \text{odd}(u) \supset f(v) < f(u)$$
$$\mu \rightarrow m$$

to the preconditions

$$\kappa,\lambda \in \mathbb{N}$$
$$\alpha(u,u)$$
$$\alpha(w,u) \wedge \neg\alpha(w,v) \supset \alpha(v,u)$$
$$\kappa \leq \lambda$$

yields

$$1,100 \in \mathbb{N}$$
$$\text{odd}(u) \supset f(u) < f(u)$$
$$\lceil \text{odd}(u) \supset f(w) < f(u) \rceil \wedge \neg \lceil \text{odd}(v) \supset f(w) < f(v) \rceil \supset \lceil \text{odd}(u) \supset f(v) < f(u) \rceil$$
$$1 < 100.$$

The first and last conditions are obviously true; the second holds since $\leq$ is reflexive; the third follows from transitivity. Applying the above instantiation mapping to the schema yields

```
R1: begin
      comment  function minimum program
      (m,i) := (1,1)
      loop  assert (∀u∈[1:i])(odd(u)⊃f(m)<f(u)), i∈ℕ
            until i=100
            i := i+1
            if odd(i)∧f(i)<f(m) then m := i fi
            repeat
      assert (∀u∈[1:100])(odd(u)⊃f(m)<f(u))
      end.
```

(The transformed conditional test $\neg(\text{odd}(i) \supset f(m) < f(i))$ simplifies to

odd(i)∧f(i)<f(m).)

Since the preconditions were satisfied by the instantiation, this program is guaranteed to be correct. Further improvements to the instantiated program can be made (e.g. incrementing i by two) by a straightforward series of program transformations such as those catalogued in Standish, et al. [1976].

Example 2: Binary-Search Schema

The following two programs both use the binary-search technique. The first program

```
P2: begin
      comment  real division program
      assert 0≤c<d, e>0
      (q,y) := (0,1)
      loop  assert q≤c/d, c/d<q+y
            until y<e
            y := y/2
            if d×(q+y)≤c then q := q+y fi
            repeat
      assert |q-c/d|<e
      end
```

uses that technique to find the quotient q of two nonnegative real numbers c and d, c<d, within a given positive tolerance e. The second program

```
Q2: begin
      comment  cube-root program
      assert x>1, t>0
      (r,w) := (1,x)
      loop  assert r<x^(1/3), x^(1/3)<w
            while w-r>t
            s := (w+r)/2
            if   s^3<x    then r := s
                          else w := s fi
            repeat
      assert |r-x^(1/3)|<t
      end
```

finds the cube root r of the real number x, x>1, within positive tolerance t.

In comparing their respective output specifications, an obvious analogy is

$$\begin{aligned} q &\leftrightarrow r \\ u/d &\leftrightarrow u^{1/3} \\ c &\leftrightarrow x \\ e &\leftrightarrow t. \end{aligned}$$

If we apply the abstraction mapping

$$\begin{aligned} q &\rightarrow \xi \\ u/d &\rightarrow \phi(u) \\ c &\rightarrow \rho \\ e &\rightarrow \varepsilon \end{aligned}$$

to $P_2$, we obtain the schema

```
S2: begin
      comment  tentative binary-search schema
      suggest 0<ρ<d, ε>0
      (ξ,y) := (0,1)
      loop  suggest ξ<φ(ρ), φ(ρ)<ξ+y
            until y<ε
            y := y/2
            if d×(ξ+y)<ρ then ξ := ξ+y fi
            repeat
      suggest |ξ-φ(ρ)|<ε
      end.
```

As in the previous example, we have replaced the assertions with suggestions to indicate that this schema may be incorrect.

This time, rather than look just at verification conditions, we imagine that the programmer has supplied detailed comments on the reasoning he employed in constructing $P_2$ and $Q_2$. Such additional information may help in extending the analogy between the programs and arriving at a correct schema. It may also allow the localization of transformations to relevant occurrences of symbols and will help avoid unnecessarily strict preconditions.

The output specification of $P_2$ was

**assert** $|q-c/d|<e$.

The programmer achieved the desired relation $|q-c/d|<e$ by decomposing it into the three conjunctive subgoals given in the

**purpose** $q<c/d$, $c/d<q+y$, $y<e$.

(This is just a comment left by the programmer listing the relations he meant his code to achieve.) The last conjunct became the exit test of the loop and the other two became loop invariants. Abstracting these subgoals, by applying the mapping, gives

$$\textbf{purpose}\ \xi<\phi(\rho),\ \phi(\rho)<\xi+y,\ y<\varepsilon,$$

which indeed imply (and are all needed to imply) the desired output specification of the schema $|\xi-\phi(\rho)|<\varepsilon$.

Similarly, the goal

$$\textbf{purpose}\ |r-x^{1/3}|<t$$

of program $Q_2$ was reduced to the three subgoals

$$\textbf{purpose}\ r<x^{1/3},\ x^{1/3}<w,\ \neg(w-r>t).$$

Applying the abstraction mapping

$$\begin{aligned} r &\rightarrow \xi \\ u^{1/3} &\rightarrow \phi(u) \\ x &\rightarrow \rho \\ t &\rightarrow \varepsilon \end{aligned}$$

to these subgoals yields

$$\textbf{purpose}\ \xi<\phi(\rho),\ \phi(\rho)<w,\ \neg(w-\xi>\varepsilon).$$

This is not however identical with the subgoals for $P_2$; to make them equivalent requires extending the analogy with

$$\xi+y \longleftrightarrow w.$$

If we let $\eta$ be their abstract counterpart, then

$$y \rightarrow \eta-\xi$$

must be added to the abstraction mapping of $P_2$ and

to the mapping for $Q_2$.

There are, however, problems with applying the transformation $y \rightarrow \eta - \xi$ to the statements of $P_2$. If we apply it in a straightforward manner to the initialization assignment $(\xi,y):=(0,1)$, then we obtain $(\xi,\eta-\xi):=(0,1)$. But the assignment $\eta-\xi:=1$ is illegal, as an expression such as $\eta-\xi$ may not appear on the left-hand side of an assignment. Nevertheless, the desired effect of making the difference between the new values of $\eta$ and $\xi$ equal to 1 can be achieved by the legal assignment $\eta:=1+0$, since 0 is the new value of $\xi$. Similarly, the transformed loop-body assignment $\eta-\xi:=(\eta-\xi)/2$ is illegal. To make it legal, the $\xi$ must be transposed to the right-hand side; the resulant assignment is $\eta:=(\eta+\xi)/2$.

We are not yet finished, however, as the value of the difference $\eta-\xi$ also changes whenever $\xi$ is assigned to. Accordingly, we must look at the then-branch assignment $\xi:=\xi+y$, or rather at $(\xi,y):=(\xi+y,y)$, where we have explicitly included a dummy assignment to the variable y. Transforming this gives $(\xi,\eta-\xi):=(\xi+\eta-\xi,\eta-\xi)$. Thus, $\xi$ should get the value $\eta$ and $\eta$ should get the value $\eta-\xi$ plus the <u>new</u> value of $\xi$, i.e. the <u>old</u> value of $\eta$. The appropriate legal assignment is accordingly $(\xi,\eta):=(\eta,2*\eta-\xi)$.

At this stage, the abstracted program is

```
S2: begin
      comment  tentative binary-search schema
      suggest 0<ρ<d, ε>0
      (ξ,η) := (0,1)
      loop  suggest ξ<φ(ρ), φ(ρ)<η
            until η-ξ<ε
            η := (η+ξ)/2
            if d×η<ρ then (ξ,η) := (η,2*η-ξ) fi
            repeat
      suggest |ξ-φ(ρ)|<ε
      end.
```

The initialization of the division program was correct since q=0 and y=1 imply $q<c/d<q+y$; the correctness of the initialization of the cube-root program follows from the fact that r=1 and w=a imply $r<a^{1/3}<w$. This suggests extending the analogy with $0 \longleftrightarrow 1$ and $1 \longleftrightarrow a$ and extending the abstraction mappings with

$$0 \rightarrow o \leftarrow 1$$
$$1 \rightarrow \imath \leftarrow x.$$

These transformations need only be applied to the respective initializations. The resultant abstract condition for the correctness of the initialization is

$$o<\phi(\rho)<\imath.$$

This is our first precondition; it replaces the abstracted input specification $0<\rho<d$.

The purpose that the programmer had in mind for the loop body of $P_2$ was

**purpose** $q<c/d,\ c/d<q+y,\ 0<y<y'$,

where $y'$ denotes the value of the variable y when control was last at the head of the loop. In other words, the loop body reachieves the invariants while making progress towards the exit test by decreasing y (some minimal amount — to ensure termination). Abstracting this goal, or the corresponding one for $Q_2$, yields

$$\textbf{purpose } \xi \leq \phi(\rho),\ \phi(\rho) < \eta,\ 0 < \eta - \xi < \eta' - \xi'.$$

To achieve the last conjunct of the loop-body goal, the division program decreases y. Then, to achieve the remaining two conjuncts, a conditional statement with the

$$\textbf{purpose } q \leq c/d,\ c/d < q+y$$

(for the decreased value of y) is introduced. Since only the value of y has so far been changed by the loop body, $q \leq c/d$ still holds, but $c/d < q+y$ might not. The program determines if the latter still holds by testing the equivalent condition $d \times (q+y) \leq c$. It is here, however, that the correctness of the schema breaks down: the transformed test $d \times \eta \leq \rho$ does not determine if the abstract relation $\phi(\rho) < \eta$ holds. Similarly, the test $\eta^3 \leq \rho$ obtained by abstracting $Q_2$ does not test for $\phi(\rho) < \eta$.

There is, nevertheless, an analogy between the tests in $P_2$ and $Q_2$. Where the former has the function $d \times u$, the latter has $u^3$. Accordingly, the analogy between $P_2$ and $Q_2$ is extended with

$$d \times u \rightarrow \phi(u) \leftarrow u^3.$$

Now, for the conditional statement to have the desired effect, we need $\neg(\phi((\eta+\xi)/2) \leq \rho)$ to imply $\phi(\rho) < (\eta+\xi)/2$, or more generally

$$\neg(\phi(u) \leq v) \supset \phi(v) < u.$$

This becomes a precondition. We must also determine the conditions under

which the then-branch of the conditional is correct. That case was correct in the division program because $d \times (q+y/2) < c$ implies $q+y/2 < c/d$ and $c/d < q+y$ implies $c/d < q+y/2+y/2$. For the abstracted schema, then, we need $\phi((\eta+\xi)/2) < \rho$ to imply $(\eta+\xi)/2 < \phi(\rho)$ and $\phi(\rho) < \eta$ to imply $\phi(\rho) < 2*(\eta+\xi)/2-\xi$. The second implication obviously holds; the first yields the precondition

$$\phi(u) < v \supset u < \phi(v).$$

Combined with the previous precondition, we have

$$\phi(u) < v \equiv u < \phi(v),$$

which holds, in particular, if $\phi$ is the inverse of a monotonic function $\phi$, i.e. if $\phi(\phi(u)) = u$ and $u < v \equiv \phi(u) < \phi(v)$.

Putting everything together, the complete abstraction mapping for $P_2$ is

$$\begin{aligned} q &\rightarrow \xi \\ u/d &\rightarrow \phi(u) \\ c &\rightarrow \rho \\ e &\rightarrow \varepsilon \\ d \times u &\rightarrow \phi(u) \\ y &\rightarrow \eta - \xi \\ 0 &\rightarrow o \\ 1 &\rightarrow \iota \end{aligned}$$

and we have derived the schema

```
S2: begin
      comment  binary-search schema
      assert ψ(u)<v≡u<φ(v), o<φ(ρ)<ι, ε>0
      (ξ,η) := (o,ι)
      loop  assert ξ<φ(ρ), φ(ρ)<η
            until η-ξ<ε
            η := (η+ξ)/2
            if ψ(η)<ρ then (ξ,η) := (η,2*η-ξ) fi
            repeat
      assert |ξ-φ(ρ)|<ε
      end.
```

This schema may be slightly optimized by applying the correctness-preserving global transformation $\eta \rightarrow \xi+\eta$. The schema

```
S2: begin
      comment  binary-search schema
      assert ψ(u)<v≡u<φ(v), o<φ(ρ)<ι, ε>0
      (ξ,η) := (o,ι-o)
      loop  assert ξ<φ(ρ), φ(ρ)<ξ+η
            until η<ε
            η := η/2
            if ψ(ξ+η)<ρ then ξ := ξ+η fi
            repeat
      assert |ξ-φ(ρ)|<ε
      end
```

that results is more similar to $P_2$. It is a general-purpose program schema that performs a binary-search for the value $\xi$ of an invertible monotonic function $\phi$ at the point $\rho$ within a tolerance $\varepsilon$.

We show now how this binary-search schema may be applied to the computation of integer square-roots. Our goal is to construct a program that sets the value of a variable z to $\lfloor\sqrt{n}\rfloor$, where $n \in \mathbb{N}$ and $\lfloor u \rfloor$ is the largest integer less than or equal to u. This attempt will illustrate some of the

difficulties that may arise in trying to apply a schema.

In this case, we cannot directly match our goal

$$\textbf{achieve } z=\lfloor\sqrt{n}\rfloor \textbf{ varying } z$$

with the output specification of the schema

$$\textbf{assert } |\xi-\phi(\rho)|<\varepsilon.$$

To bring out the analogy we must first expand the goal $z=\lfloor\sqrt{n}\rfloor$ using the definition of $\lfloor u\rfloor$, giving the equivalent goal

$$\textbf{achieve } z<\sqrt{n},\ \sqrt{n}<z+1,\ z\varepsilon\mathbf{Z}.$$

Since we know that the schema also achieves the two output invariants

$$\textbf{assert } \xi<\phi(\rho),\ \phi(\rho)<\xi+\varepsilon,$$

we can compare these invariants with our goal. This suggests the instantiation mapping

$$\begin{aligned}\xi &\rightarrow z\\ \phi &\rightarrow \sqrt{\ }\\ \rho &\rightarrow n\\ \varepsilon &\rightarrow 1\end{aligned}$$

to achieve the first two conjuncts of the goal. In addition we will have to somehow achieve $z\varepsilon\mathbf{Z}$.

The preconditions for the schema's correctness are

$$\textbf{assert } \phi(u)<v\equiv u<\phi(v),\ 0<\phi(\rho)<1;$$

instantiating them yields

$$\phi(u)<v \equiv u<\sqrt{v},\quad o<\sqrt{n}<\iota .$$

Since $u^2<v$ is equivalent to $u<\sqrt{v}$ when $u>0$, the first condition may be satisfied by taking $\phi(u)$ to be $u^2$, provided that the argument u is never negative. Noting that $\sqrt{n}>0$ suggests letting $o=0$; $\sqrt{n}<\iota$ can be satisfied by letting $\iota=n+1$. This all suggests completing the instantiation to obtain

$$\begin{aligned} \xi &\rightarrow z \\ \phi &\rightarrow \sqrt{\ } \\ \rho &\rightarrow n \\ \varepsilon &\rightarrow 1 \\ \phi(u) &\rightarrow u^2 \\ o &\rightarrow 0 \\ \iota &\rightarrow n+1. \end{aligned}$$

Applying this instantiation to the schema, we obtain the program

```
R2: begin
      comment  instantiated schema
      assert nε𝕀N
      (z,η) := (0,n+1)
      loop  assert z<√n, √n<z+η
            until η<1
            η := η/2
            if (z+η)^2<n then z := z+η fi
            repeat
      assert z<√n, √n<z+1
      end.
```

It is easy to establish that both $\eta$ and z are nonnegative; thus, the conditional test $\eta^2<n$ is equivalent to $\eta<\sqrt{n}$, as required.

We are still left with achieving the subgoal $z\varepsilon\mathbf{Z}$. One way that this may be done is by perturbing the final value of z just enough to make it an integer while preserving the two relations that the instantiated schema

achieves. Accordingly, we append the statement

```
if   n<(⌊z⌋+1)^2   then z := ⌊z⌋
                   else z := ⌊z⌋+1 fi.
```

Alternatively, $z \varepsilon \mathbf{Z}$ could be achieved by ensuring that the assignments to z preserve that relation. This avenue is pursued in Dershowitz [1978], where a series of additional transformations results in an improved version of integer square-root.

Our final program is

```
R2: begin
      comment  integer square-root program
      assert nεℕ
      (z,η) := (0,n+1)
      loop  assert z<√n, √n<z+η
            until η<1
            η := η/2
            if (z+η)^2<n then z := z+η fi
            repeat
      if   n<(⌊z⌋+1)^2   then z := ⌊z⌋
                         else z := ⌊z⌋+1 fi
      assert z=⌊√n⌋
      end.
```

Example 3: Associative Recursion Schema

Abstraction may also be used for the design of correctness-preserving program transformations. For example, one can first write iterative versions of several recursive programs; by abstracting those programs, a more general recursion-to-iteration transformation schema can be obtained. As we shall see in the following example, very little information is gleaned by a comparison of output specifications; instead, most of the analogy is derived from the verification conditions.

The two given programs are

```
P3: begin
      comment  factorial program
      assert nεℕ
      (z,y) := (1,n)
      loop  assert y!×z=n!, yεℕ
            until y=0
            (z,y) := (y×z,y-1)
            repeat
      assert z=n!
      end
```

and

```
Q3: begin
      comment summation program
      assert k,m∈Z, k≤m
      (s,j) := (0,k)
      loop  assert Σ_{i=j}^{m} f(i)+s=Σ_{i=k}^{m} f(i), j∈Z
            until j=m+1
            (s,j) := (f(j)+s,j+1)
            repeat
      assert s=Σ_{i=k}^{m} f(i)
      end.
```

Our object is to derive a schema for computing recursive functions that are like factorial and summation.

Matching the two output specifications

$$\textbf{assert } z=n!$$

and

$$\textbf{assert } s=\Sigma_{i=k}^{m} f(i)$$

suggests as one possible analogy

$$z \longleftrightarrow s$$
$$n \longleftrightarrow k$$
$$u! \longleftrightarrow \Sigma_{i=u}^{m} f(i).$$

The two output variables z and s generalize to the abstract output variable $\zeta$; the input variables n and k generalize to $\chi$; the two functions u! and $\Sigma_{i=u}^{m} f(i)$ generalize to an abstract function variable $\theta(u)$.

Applying the abstraction mapping

$$z \rightarrow \zeta$$
$$n \rightarrow \chi$$
$$u! \rightarrow \theta(u)$$

to $P_3$ yields

```
S3: begin
      comment  abstracted factorial program
      suggest χεℕ
      (ζ,y) := (1,χ)
      loop  suggest θ(y)×ζ=θ(χ), yεℕ
            until y=0
            (ζ,y) := (y×ζ,y-1)
            repeat
      suggest ζ=θ(χ)
      end.
```

Applying the mapping

$$s \rightarrow \zeta$$
$$k \rightarrow \chi$$
$$\Sigma_{i=u}^{m} f(i) \rightarrow \theta(u)$$

to $Q_3$ yields a schema with the same output suggestion:

```
T3: begin
      comment  abstracted summation program
      suggest χ,mεℤ, χ≤m
      (ζ,j) := (0,χ)
      loop  suggest θ(j)+ζ=θ(χ), jεℤ
            until j=m+1
            (ζ,j) := (f(j)+ζ,j+1)
            repeat
      suggest ζ=θ(χ)
      end.
```

Neither abstract program is, however, correct.

Both programs consist of a single loop; their respective abstracted loop invariants are

$$\textbf{assert } \theta(y)\times\zeta=\theta(\chi),\ y\varepsilon\mathbb{N}$$

and

$$\textbf{assert } \theta(j)+\zeta=\theta(\chi),\ j\varepsilon\mathbf{Z}.$$

Matching the invariants suggests the additional aspect of the analogy

$$\begin{aligned} y &\rightarrow \nu \leftarrow j \\ \times &\rightarrow \tau \leftarrow + \\ \mathbb{N} &\rightarrow \Omega \leftarrow \mathbf{Z}. \end{aligned}$$

Applying the corresponding transformations to the loop invariants we obtain the abstract invariant

$$\textbf{assert } \tau(\theta(\nu),\zeta)=\theta(\chi),\ \nu\varepsilon\Omega.$$

Now, we must consider the verification conditions. Applying the transformations to the initialization condition of $P_3$ we get

$$\begin{aligned} &\tau(\theta(\chi),1)=\theta(\chi) \\ &\chi\varepsilon\Omega; \end{aligned}$$

on the other hand, applying the transformations to the initialization condition of $Q_3$ gives

$$\begin{aligned} &\tau(\theta(\chi),0)=\theta(\chi) \\ &\chi\varepsilon\Omega. \end{aligned}$$

The shared condition

$$\chi \varepsilon \Omega$$

becomes a precondition for the schema. To unify the remaining conditions of the two programs we add to the analogy

$$1 \rightarrow \omega \leftarrow 0,$$

to obtain the abstract condition

$$\tau(\theta(\chi),\omega)=\theta(\chi),$$

or more generally,

$$\tau(u,\omega)=u.$$

This too is a precondition.

The loop-exit condition derived from $P_3$ is

$$\tau(\theta(\nu),\zeta)=\theta(\chi) \wedge \nu \varepsilon \Omega \wedge \nu=0 \supset \zeta=\theta(\chi);$$

from $Q_3$, we get

$$\tau(\theta(\nu),\zeta)=\theta(\chi) \wedge \nu \varepsilon \Omega \wedge \nu=m+1 \supset \zeta=\theta(\chi).$$

With the additional abstraction

$$0 \rightarrow \gamma \leftarrow m+1$$

we get

$$\tau(\theta(\nu),\zeta)=\theta(\chi) \wedge \nu \varepsilon \Omega \wedge \nu=\gamma \supset \zeta=\theta(\chi).$$

For simplicity, we shall replace this condition with the stronger

$$\tau(\theta(\gamma),u)=u.$$

For the loop-body paths, we have the conditions

$$\tau(\theta(\nu),\zeta)=\theta(\chi) \wedge \nu\varepsilon\Omega \wedge \nu\neq\gamma \supset \tau(\theta(\nu-1),\tau(\nu,\zeta))=\theta(\chi) \wedge \nu-1\varepsilon\Omega$$

and

$$\tau(\theta(\nu),\zeta)=\theta(\chi) \wedge \nu\varepsilon\Omega \wedge \nu\neq\gamma \supset \tau(\theta(\nu+1),\tau(f(\nu),\zeta))=\theta(\chi) \wedge \nu+1\varepsilon\Omega,$$

for $P_3$ and $Q_3$, respectively. To unify $\nu-1\longleftrightarrow\nu+1$ and $\nu\longleftrightarrow f(\nu)$, we need the additional abstractions

$$u-1 \rightarrow \delta(u) \leftarrow u+1$$
$$\nu \rightarrow \sigma(\nu) \leftarrow f(\nu).$$

(The transformation $\nu\rightarrow\sigma(\nu)$ applied to all occurrences of $\nu$ in $P_3$ would be overzealous; it should be localized to the occurrence that led to the difference between the two verification conditions. That occurrence is in the assignment $\xi:=\tau(\nu,\xi)$ obtained from the original $z:=y\times z$.) This abstraction mapping yields

$$\tau(\theta(\nu),\zeta)=\theta(\chi) \wedge \nu\varepsilon\Omega \wedge \nu\neq\gamma \supset \tau(\theta(\delta(\nu)),\tau(\sigma(\nu),\zeta))=\theta(\chi) \wedge \delta(\nu)\varepsilon\Omega,$$

for which we shall use the two preconditions

$$\nu\neq\gamma \supset \tau(\theta(\nu),u)=\tau(\theta(\delta(\nu)),\tau(\sigma(\nu),u))$$
$$\nu\varepsilon\Omega \wedge \nu\neq\gamma \supset \delta(\nu)\varepsilon\Omega.$$

Finally, the verification condition for the termination of the abstracted program is

$$(\exists u\varepsilon\mathbb{N})\delta^u(\chi)=\gamma.$$

This too becomes a precondition.

The complete abstraction is

$$
\begin{aligned}
z &\rightarrow \zeta \leftarrow s \\
n &\rightarrow \chi \leftarrow k \\
u! &\rightarrow \theta(u) \leftarrow \Sigma_{i=u}^{m} f(i) \\
y &\rightarrow \nu \leftarrow j \\
\times &\rightarrow \tau \leftarrow + \\
\mathbb{N} &\rightarrow \Omega \leftarrow \mathbf{Z} \\
1 &\rightarrow \omega \leftarrow 0 \\
0 &\rightarrow \gamma \leftarrow m+1 \\
u-1 &\rightarrow \delta(u) \leftarrow u+1 \\
\nu &\rightarrow \sigma(\nu) \leftarrow f(\nu).
\end{aligned}
$$

Applying these abstraction transformations (in order) to $Q_3$, and collecting all the preconditions, we derive the schema

$S_3$: **begin**
**comment** associative recursion schema
**assert** $\tau(u,\omega)=u$, $\tau(\theta(\gamma),u)=u$,
$\nu\neq\gamma \supset \tau(\theta(\nu),u)=\tau(\theta(\delta(\nu)),\tau(\sigma(\nu),u))$,
$\chi\in\Omega$, $\nu\in\Omega \wedge \nu\neq\gamma \supset \delta(\nu)\in\Omega$,
$(\exists u\in\mathbb{N})\delta^{u}(\chi)=\gamma$
$(\zeta,\nu) := (\omega,\chi)$
**loop** **assert** $\tau(\theta(\nu),\zeta)=\theta(\chi)$, $\nu\in\Omega$
**until** $\nu=\gamma$
$(\zeta,\nu) := (\tau(\sigma(\nu),\zeta),\delta(\nu))$
**repeat**
**assert** $\zeta=\theta(\chi)$
**end.**

In this manner we have obtained a general schema for computing a function $\theta(\chi)$. It applies to recursive functions $\theta(\chi)$ such that $\theta(\gamma)=\omega$ is a unit of an associative and commutative function $\tau$, and $\theta(u)=\tau(\theta(\delta(u)),\sigma(u))$

when $u \neq \gamma$. The schema is similar to one of the recursion-to-iteration transformations of Darlington and Burstall [1978].

To see how this schema may be applied to another problem, consider the specifications

$R_3$: **begin**
**comment** list-reversal program
**assert** $\ell \varepsilon L$
**achieve** z=reverse($\ell$) **varying** z
**end**,

where L is a set of lists and reverse($\ell$) is the list containing the elements of $\ell$ in reverse order. Assume that we are also given two relevant facts about reverse: reverse(())=(), where () is the empty list, and reverse(u)=reverse(tail(u))*(head(u)) when u≠(), where u*v concatenates the two lists u and v, (head(u)) is a one element list containing the first element of u, and tail(u) is a list of all but the first element.

An initial comparison of the schema's output specification $\zeta=\theta(\chi)$ with the new specification z=reverse($\ell$) suggests the instantiation

$$\theta \rightarrow \text{reverse}$$
$$\zeta \rightarrow z$$
$$\chi \rightarrow \ell.$$

Instantiating the precondition

$$v \neq \gamma \supset \tau(\theta(v),u)=\tau(\theta(\delta(v)),\tau(\sigma(v),u))$$

gives

$$v \neq \gamma \supset \tau(\text{reverse}(v),u)=\tau(\text{reverse}(\delta(v)),\tau(\sigma(v),u)).$$

By the second of the above two facts, we have that reverse(v) may be replaced by reverse(tail(v))*(head(v)), provided that v is not the empty list (). This

suggests the possibility of instantiating $\gamma \rightarrow ()$ to obtain

$$v \neq () \supset \tau(\text{reverse}(\text{tail}(v))*(\text{head}(v)),u)=\tau(\text{reverse}(\delta(v)),\tau(\sigma(v),u)).$$

The function reverse appears on the two sides of the equality, so we try to generalize this condition by replacing both occurrences of reverse with an arbitrary list w. (This is similar to the generalization technique used in the theorem prover described in Boyer and Moore [1980].) To do that, we must first unify reverse($\delta$(v)) with reverse(tail(v)) by instantiating $\delta \rightarrow$tail. We are left with

$$v \neq () \supset \tau(w*(\text{head}(v)),u)=\tau(w,\tau(\sigma(v),u)).$$

Similarly, we unify $\sigma$(v) with (head(v)), the list containing just the first element of v, obtaining

$$v \neq () \supset \tau(w*v,u)=\tau(w,\tau(v,u)).$$

This matches with the fact that * is associative, i.e. $(w*v)*u=w*(v*u)$, by instantiating $\tau \rightarrow *$.

Applying the instantiations that we have found to the other five preconditions

$$\tau(u,\omega)=u$$
$$\tau(\theta(\gamma),u)=u$$
$$\chi \in \Omega$$
$$v \in \Omega \wedge v \neq \gamma \supset \delta(v) \in \Omega$$
$$(\exists u \in \mathbb{N})\delta^{u}(\chi)=\gamma$$

yields

$$u*\omega=u$$
$$\text{reverse}(())*u=u$$
$$\ell\varepsilon\Omega$$
$$v\varepsilon\Omega \wedge v\neq() \supset \text{tail}(v)\varepsilon\Omega$$
$$(\exists u\varepsilon\mathbb{N})\text{tail}^{u}(\ell)=().$$

But reverse(())=() and ()*u=u, since the empty list () is an identity element of the function *. Thus, the second condition holds. The first suggests letting $\omega\rightarrow()$; the third and fourth suggest letting $\Omega\rightarrow L$. The last condition then follows from the third, as a property of lists.

The completed instantiation is

$$\theta \rightarrow \text{reverse}$$
$$\zeta \rightarrow z$$
$$\chi \rightarrow r$$
$$\gamma \rightarrow ()$$
$$\delta \rightarrow \text{tail}$$
$$\sigma(u) \rightarrow (\text{head}(u))$$
$$\tau \rightarrow *$$
$$\omega \rightarrow ().$$

In all, we have derived the following program

```
R3: begin
      comment  list reversal program
      assert ℓεL
      (z,v) := ((),ℓ)
      loop  assert reverse(v)*z=reverse(ℓ), vεL
            until v=()
            (z,v) := ((head(v))*z,tail(v))
            repeat
      assert z=reverse(ℓ)
      end.
```

In this example most of the instantiation mapping was derived from an analysis of the preconditions. The preconditions served, in this way, to guide the construction of the list-reversal program in the pattern of other recursive functions.

## III. DISCUSSION

We have presented several examples demonstrating a methodology for deriving an abstract program schema that captures the technique underlying a given set of concrete programs. Once derived, the schema may be applied to solve new problems by instantiating the abstract entitites of the schema with concrete elements from the problem domain. We have also seen how this methodology can be used to derive correctness-preserving program transformations and to guide their application.

Abstraction and instantiation complement techniques of program transformation, such as have been advocated by Knuth [1974]. When faced with the task of developing a new program (or subprogram) to meet a set of specifications, the programmer ought to first search for an applicable schema. After instantiating the schema, transformations may be applied to solve any remaining specifications or to increase efficiency. If no applicable schema can be found, one might still be able to find a schema or program solving some analogous problem, and modify it (see, for example, Dershowitz and Manna [1977]). The two programs together would then be used to formulate a schema for future use.

There are a few problems inherent in the use of analogies for program abstraction and instantiation. These include "hidden" analogies, "misleading" analogies, "incomplete" analogies, and "overzealous" analogies.

Hidden analogies arise when given specifications (of the two or more existing programs in the case of abstraction, and of the abstract schema and concrete problem in the case of instantiation) that are to be compared with one another have little syntactically in common. Since the pattern-matching ideas that we have employed are syntax based, when the specifications are not syntactically similar, the underlying analogy would be hidden. In such a situation it is necessary to rephrase the specifications in some equivalent manner that brings their similarity out, before an analogy can be found. This is clearly a difficult problem in its own right; in general some form of means-end analysis seems appropriate.

At the opposite extreme, a syntactic analogy may be misleading. The same symbol may appear in the specifications of two programs, yet may play nonanalogous roles in the two programs. Two programs might even have the exact same specifications, but employ totally different methods of solution. Situations such as these would be detected in the course of analyzing the correctness conditions for the abstracted programs.

We have seen how the proof of correctness of a program can be used to help avoid overzealously applying transformations to unrelated parts of a program. The proof also helps complete an analogy between two programs, only part of which was found by a comparison of specifications.

The methods we have described appear to be amenable to automation. The necessary reasoning ability is the same as is needed for a program-verification system; the program manipulation abilities are similar to what is required of program-transformation systems. Of course, we do not expect these methods alone to suffice for an automatic program-abstraction system to produce and apply program schemata. But we can envision the possibility of such methods being embedded in a semi-automatic program-development environment in which the system performs the more straightforward steps, and the human programmer guides the machine in the more creative ones.

ACKNOWLEDGMENT

I sincerely thank Zohar Manna and Richard Waldinger for their guidance in initiating this research.

## APPENDIX

Gerhart [1975] recommended the hand-compilation of a handbook of program schemata. Such a collection of shemata, together with a library of program transformations, could be used in an interactive program-development system. In this appendix, we have culled fifteen representative schemata from the programming literature.

We use the following nomenclature:

| | |
|---|---|
| $\mathbb{N}$ | set of natural numbers |
| $\mathbb{R}$ | set of real numbers |
| $\mathbb{Z}$ | set of integers |
| [u:v] | set of integers between u and v |
| j,k,t | input variable |
| x | input variable or vector of variables |
| p,q | predicate symbol |
| c,d,e,f,g,h | function symbol |
| a,b | constant symbol |
| z,r | output variable |
| i,s,y,m | program variable |
| u,v,w | universally quantified variable (quantifier often omitted) |
| n | existentially quantified variable |

Each of the following schemata is followed by an output assertion giving its abstract input-output specification. They are preceded by an input assertion containing the preconditions for correct application. The general format is

```
S_i: begin
     comment  title (source)
          purpose
     assert  type conditions,
          correctness preconditions,
          termination precondition
     ...
     schema body
     ...
     assert  output specification
     end.
```

The references are to sources that present an abstract schema; they are somewhat arbitrary, as the methods themselves are all well known. (The schemata may differ in details from those given in the referenced sources.)

```
S_1: begin
     comment element-by-element (Gerhart [1975])
          achieve q for all integers between j and k
     assert jεℤ, kεℝ, bεx,
          uε[j:k]∧vεx ⊃ h(u,v)εx,
          uε[j:k]∧vεx ⊃ q(h(u,v),u),
          u,wε[j:k]∧vεx∧u<w∧q(v,u) ⊃ q(h(w,v),u)
     (z,i) := (b,j)
     loop  assert (∀uε[j:i-1])q(z,u), iε[j:k+1]∨i=j>k+1, zεx
          until i>k
          (z,i) := (h(i,z),i+1)
          repeat
     assert (∀uε[j:k])q(z,u), zεx
     end
```

```
S2: begin
      comment extremum (Dershowitz and Manna [1975])
            achieve q for all integers between j and k
      assert jεℤ, kεℝ,
            uε[j:k] ⊃ q(u,u),
            u,v,wε[j:k]∧u<v∧q(w,u)∧¬q(w,v) ⊃ q(v,u)
      (z,i) := (j,j)
      loop  assert (∀uε[j:i])q(z,u), z,iε[j:k]∨i=j>k
            until i>k-1
            i := i+1
            if ¬q(z,i) then z := i fi
            repeat
      assert (∀uε[j:k])q(z,u), zε[j:k]∨j>k
      end
```

```
S3: begin
      comment linear-search (Dijkstra [1972])
            find least integer between j and k such that q holds
      assert jεℤ, kεℝ
      z := j
      loop  assert (∀uε[j:z-1])¬q(u), zε[j:k+1]∨z=j>k+1
            until z>k∨q(z)
            z := z+1
            repeat
      assert (∃nε[j:k])q(n) ⊃ z=(min nε[j:k])q(n),
            z>j, z>k∨q(z)
      end
```

```
S4: begin
    comment gradient-search (Misra [1978])
          search for local extremum
    assert aεx,
          uεx ⊃ t(u) ⊑ x,
          uεx ⊃ q(u,u),
          u,v,wεx∧q(w,u)∧¬q(w,v) ⊃ q(v,u),
          u⊑x ⊃ g(u)εu,
          u,v,wεx∧q(u,w) ⊃ q(u,v)∨q(v,w),
          (∃nε𝕀N)|x|=n
    comment  |x| denotes the number of elements in the set x
    z := a
    loop  (m,s) := (z,t(z))
          loop  assert (∀uεt(m)-s)q(z,u), z,mεx, s⊑x
                until s={}
                y := g(s)
                s := s-{y}
                if ¬q(z,y) then z := y fi
                repeat
          assert (∀uεt(m))q(z,u), zεx
          until z=m
          repeat
    assert (∀uεt(z))q(z,u), zεx
    end
```

```
S5: begin
      comment binary-search (Dershowitz and Manna [1975])
            approximate transition from q true to q false
      assert a,b,t∈ℝ,
            q(a), ¬q(b),
            q(u+t)∧t>v ⊃ q(u+v),
            t>0
      (z,y) := (a,b-a)
      loop  assert q(z), ¬q(z+y), a≤z≤z+y≤b∨a>b
            until y≤t
            y := y/2
            if q(z+y) then z := z+y fi
            repeat
      assert q(z), ¬q(z+t), a≤z<b∨a>b
      end
```

```
S6: begin proc sort(x[j:k])
    comment sorting (Darlington [1978])
          sort array segment x[j:k]
    assert x[j:k]εa[j:k],
          uεa[u´:w´] ⊃ f(u)εa[u´:w´]×[u´:w´-1],
          uεa[u´:v´]∧vεa[v´+1:w´] ⊃ g(u,v)εa[u´:w´],
          uεa[u´:v´]∧vεa[v´+1:w´] ⊃ bag(g(u,v))=bag(u)⊔bag(v),
          uεa[u´:w´]∧f(u)=(w,v) ⊃ bag(u)=bag(w),
          uεa[u´:w´]∧f(u)=(w[u´:w´],v)∧sorted(w[u´:v])∧sorted(w[v+1:w´]) ⊃
                sorted(g(w[u´:v],w[v+1:w´]))
    comment bag(u)=bag(v) means that for each occurrence
          of an element in u there is an occurrence of the element in v;
          sorted(w[u´:w´])≡(∀vε[u´:w´-1])w[v]≤w[v+1];
          a[u:v] is the set of arrays of elements of a
          with indices in [u:v];
          x̄ denotes the value of x upon procedure entry
    if j<k then (x[j:k],m) := f(x[j:k])
                sort(x[j:m])
                sort(x[m+1:k])
                x[j:k] := g(x[j:m],x[m+1:k]) fi
    assert bag(x̄[j:k])=bag(x[j:k]), sorted(x[j:k])
    end
```

```
S7: begin
    comment associative-recursion (Darlington and Burstall [1976])
          compute f(x)
    assert p(u) ⊃ f(u)=c(u),
          h(u,c(u))=u,
          ¬p(u) ⊃ f(u)=h(f(e(u)),d(u)),
          h(u,h(v,w))=h(h(u,v),w),
          (∃n∈ℕ)p(e^n(x))
    if p(x)
          then z := c(x)
          else (z,y) := (d(x),e(x))
               loop  assert h(f(y),z)=f(x)
                     until p(y)
                     (z,y) := (h(d(y),z),e(y))
                     repeat
               z := h(c(y),z)
          fi
    assert z=f(x)
    end
```

```
S8: begin
    comment tail-recursion (Wirth [1976])
          compute f(x)
    assert p(u) ⊃ f(u)=c(u),
          ¬p(u) ⊃ f(u)=f(e(u)),
          (∃n∈ℕ)p(e^n(x))
    y := x
    loop  assert f(y)=f(x)
          until p(y)
          y := e(y)
          repeat
    z := c(y)
    assert z=f(x)
    end
```

$S_9$: **begin**

**comment** right-commutative-recursion (Cooper [1966])

**assert** $p(u) \supset f(u)=a$,

$\neg p(u) \supset f(u)=h(f(e(u)),d(u))$,

$h(g(u,v),w)=g(h(u,w),v)$,

$h(a,u)=g(a,u)$,

$(\exists n\in\mathbb{N})p(e^n(x))$

$(z,y) := (a,x)$

**loop** **assert** $u=(\min\ n\in\mathbb{N})p(e^n(y))$

$\supset f(x)=g(g(\cdots g(g(z,d(y)),d(e(y))),\cdots),d(e^{u-1}(y)))$

**until** $p(y)$

$(z,y) := (g(z,d(y)),e(y))$

**repeat**

**assert** $z=f(x)$

**end**

$S_{10}$: **begin**

**comment** invertible-recursion (Cooper [1966])

compute $f(x)$

**assert** $f(a)=c(a)$,

$u\neq a \supset f(u)=h(f(e(u)),u)$,

$e(g(u))=u$,

$(\exists n\in\mathbb{N})g^n(a)=x$

$(z,y) := (c(a),a)$

**loop** **assert** $z=f(y)$, $(\exists n\in\mathbb{N})(y=g^n(a)\wedge(\forall u\in[0:n-1])g^u(a)\neq x)$

**until** $y=x$

$(z,y) := (h(z,g(y)),g(y))$

**repeat**

**assert** $z=f(x)$

**end**

```
S11: begin
     comment stack-introduction (Wossner, et al. [1978])
          compute recursive function f(x) using stack
     assert p(u) ⊃ f(u)=c(u),
          ⌝p(u) ⊃ f(u)=h(f(d(u)),u),
          (∃n∈ℕ)p(d^n(x))
     comment s is the list (s_n,s_{n-1},···,s_1);
          head(s)=s_n; tail(s)=(s_{n-1},···,s_1);
          u∘s=(u,s_n,···,s_1)
     (y,s) := (x,())
     loop  assert f(x)=h(···h(h(f(y),s_n),s_{n-1})···,s_1)
          until p(y)
          (y,s) := (d(y),y∘s)
          repeat
     z := c(y)
     loop  assert f(x)=h(···h(h(z,s_n),s_{n-1})···,s_1)
          until s=()
          (z,s) := (h(z,head(s)),tail(s))
          repeat
     assert z=f(x)
     end
```

$S_{12}$: **begin**

**comment** double-recursion (Knuth [1974])

compute f(x)

**assert** p(u) $\supset$ f(u)=c(u),

$\neg$p(u) $\supset$ f(u)=h(f(e(u)),f(d(u))),

h(a,u)=u,

h(u,h(v,w))=h(h(u,v),w),

$p(x)\vee(\exists n\in\mathbb{N})g^n(x)=\{\}$,

where $g(u)=\{e(v)\mid v\in u\wedge\neg p(v)\}\sqcup\{d(v)\mid v\in u\wedge\neg p(v)\}$

**comment** s is the list $(s_n,s_{n-1},\cdots,s_1)$;

$head(s)=s_n$; $tail(s)=(s_{n-1},\cdots,s_1)$;

$u\circ s=(u,s_n,\cdots,s_1)$

(z,s) := (a,(x))

**loop** **assert** $f(x)=h(z,h(s_1,h(s_2,h(\cdots,h(s_{n-1},s_n)\cdots))))$

**until** s=()

(y,s) := (head(s),tail(s))

**if** p(y) **then** z := h(z,c(y))

**else** s := e(y)$\circ$d(y)$\circ$s **fi**

**repeat**

**assert** z=f(x)

**end**

```
S13: begin proc back(t,x,z)
     comment backtracking (Gerhart and Yelowitz [1976])
           collect all vectors beginning with x satisfying p
     assert xεa, z⊑a,
           uεa ⊃ t(u)⊑a,
           u⊑a ⊃ g(u)εu,
           (∃nε𝕀N)|t*({x})|=n
     comment t*(u)=u⊔t̄(u)⊔t̄²(u)⊔···,
             where t̄(u)=⊔ t(v);
                        vεu
           |t*(u)| denotes the number of elements in the set t*(u);
           z̄ denotes the value of z upon procedure entry
     if p(x) then z := z⊔{x} fi
     y := t(x)
     loop  assert z̄⊔{uεt*(x)|p(u)}
                 =z⊔{uεt*(y)|p(u)}
           until y={}
           m := g(y)
           y := y-{m}
           back(t,m,z)
           repeat
     assert z=z̄⊔{uεt*(x)|p(u)}
     end
```

$S_{14}$: **begin**

**comment** marking (Yelowitz and Duncan [1977])

collect elements related by t to x

**assert** $x \sqsubseteq a$, $t \sqsubseteq a^2$,

$\{n \mid (u,n) \in w\} \sqsubseteq f(u,v,w) \sqsubseteq v$,

$u \sqsubseteq a \supset g(u) \in u$,

$(\exists n \in \mathbb{N}) \mid t^*(x) \mid = n$

**comment** $t^*(x) = x \sqcup \bar{t}(x) \sqcup \bar{t}^2(x) \sqcup \cdots$,

where $\bar{t}(x) = \bigsqcup_{u \in x} \{n \mid (u,n) \in t\}$;

$|t^*(x)|$ denotes the number of elements in the set $t^*(x)$

$(z,s,y) := (x,x,t)$

**loop** **assert** $s \sqsubseteq z \sqsubseteq t^*(x) \sqsubseteq z \sqcup y^*(s) \sqsubseteq a$, $y \sqsubseteq t$

**until** $y = \{\} \vee s = \{\}$

$m := g(s)$

$s := s \sqcup f(m,z,y) - \{m\}$

$z := z \sqcup \{u \mid (m,u) \in y\}$

$y := y - \{(u,v) \mid (m,v) \in y\}$

**repeat**

**assert** $z = t^*(x)$

**end**

```
S15: begin
     comment additive-relation (Dershowitz and Manna [1981])
          achieve p(z) maintaining relation between r and z
     assert g(u,v)=g(v,u),
          g(g(u,v),w)=g(u,g(v,w)),
          h(h(u,v),w)=h(h(u,w),v),
          h(g(u,v),w)=g(h(u,w),h(v,w)),
          (∃n∈ℕ)p(d^n(x)), where d(u)=g(u,h(f(u),a))
     (z,r) := (x,t)
     loop  assert g(h(x,b),h(r,a))=g(h(t,a),h(z,b))
          until p(z)
          (z,r) := (g(z,h(f(z),a)),g(r,h(f(z),b)))
          repeat
     assert p(z), g(h(x,b),h(r,a))=g(h(t,a),h(z,b))
     end
```

## REFERENCES

[1] R.S. Boyer and J S. Moore [1980], Computational theorem proving, Academic Press, New York.

[2] R. Brown [Oct. 1976], Reasoning by analogy, Working paper 132, Artificial Intelligence Laboratory, MIT, Cambridge, MA.

[3] T.W. Chen and N.V. Findler [Dec. 1976], Toward analogical reasoning in problem solving by computers, Technical report 115, Dept. of Computer Science, State Univ. of New York, Buffalo, NY.

[4] D.C. Cooper [May 1966], The equivalence of certain computations, Computer J., vol. 9, no. 1, pp. 45-52.

[5] J. Darlington [Dec. 1978], A synthesis of several sorting algorithms, Acta Informatica, vol. 11, no. 1, pp. 1-30.

[6] J. Darlington and R.M. Burstall [Mar. 1976], A system which automatically improves programs, Acta Informatica, vol. 6, no. 1, pp. 41-60.

[7] N. Dershowitz [Nov. 1978], The evolution of programs, Ph.D. thesis, Dept. of Applied Mathematics, Weizmann Inst. of Science, Rehovot, Israel; available as Report R-80-1017, Dept. of Computer Science, Univ. of Illinois, Urbana, IL.

[8] N. Dershowitz and Z. Manna [July 1975], On automating structured programming, Proc. Colloq. IRIA on Proving and Improving Programs, Arc-et-Senans, France, pp. 167-193.

[9] N. Dershowitz and Z. Manna [Nov. 1977], The evolution of programs: Automatic program modification, IEEE Software Engineering, vol. SE-3, no. 6, pp. 377-385.

[10] N. Dershowitz and Z. Manna [Mar. 1981], Inference rules for program annotation, IEEE Software Engineering, vol. SE-7, no. 2, pp. 207-222.

[11] P. Deussen [July 1979], One abstract parsing algorithm for all kinds of parsers, Proc. Sixth EATCS Intl. Colloq. on Automata, Languages and Programming, Graz, Austria, pp. 203-217.

[12] E.W. Dijkstra [1972], Notes on structured programming, in O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, "Structured Programming," Academic Press, London.

[13] E.W. Dijkstra [1976], A discipline of programming, Prentice-Hall, Englewood Cliffs, NJ.

[14] A.G. Duncan and L. Yelowitz [July 1979], Studies in abstract/concrete mappings in proving algorithm correctness, Proc. Sixth EATCS Intl. Colloq. on Automata, Languages and Programming, Graz, Austria, pp. 218-229.

[15] S.L. Gerhart [Apr. 1975], Knowledge about programs: A model and case study, Proc. Intl. Conf. on Reliable Software, Los Angeles, CA, pp. 88-95.

[16] S.L. Gerhart and L. Yelowitz [Dec. 1976], Control structure abstractions of the backtracking programming technique, IEEE Software Engineering, vol. SE-2, no. 4, pp. 285-292.

[17] G. Huet and B. Lang [Dec. 1978], Proving and applying program transformations expressed with second-order patterns, Acta Informatica, vol. 11, no. 1, pp. 31-55.

[18] R.E. Kling [Aug. 1971], Reasoning by analogy with applications to heuristic problem solving: A case study, Ph.D. thesis, Dept. of Computer Science, Stanford Univ., Stanford, CA.

[19] D.E. Knuth [Dec. 1974], Structured programming with goto statements, Computing Surveys, vol. 6, no. 4, pp. 261-301.

[20] S. Lee, W.P. deRoever, and S.L. Gerhart [Jan. 1979], The evolution of list-copying algorithms, Proc. Sixth ACM Symp. on Principles of Programming Languages, San Antonio, TX, pp. 53-67.

[21] J. McDermott [Aug. 1979], Learning to use analogies, Proc. Sixth Intl. Joint Conf. on Artificial Intelligence, Tokyo, Japan, pp. 568-576.

[22] Z. Manna and R.J. Waldinger [Summer 1975], Reasoning and knowledge in program synthesis, Artificial Intelligence, vol. 6, no. 2, pp. 175-208.

[23] J. Misra [Sept. 1978], An approach to formal definitions and proofs of programming principles, IEEE Software Engineering, vol. SE-4, no. 5, pp. 410-413.

[24] D.A. Plaisted [July 1980], Abstraction mappings in mechanical theorem proving, Proc. Fifth Conf. on Automated Deduction, Les Arcs, France, pp. 264-280.

[25] T.A. Standish, D.C. Harriman, D.F. Kibler, and J.M. Neighbors [Jan. 1976], The Irvine program transformation catalogue: A stock of ideas for improving programs using source-to-source transformations, Memo, Dept. of Information and Computer Sciences, Univ. of California, Irvine, CA.

[26] R.J. Sternberg [1977], Intelligence, information processing, and analogical reasoning, Lawrence Erlbaum, Hillsdale, NJ.

[27] G.J. Sussman [1975], A computer model of skill acquisition, American Elsevier, New York.

[28] J.W. Ulrich and R. Moll [Aug. 1977], Program synthesis by analogy, Proc. ACM Symp. on Artificial Intelligence and Programming Languages, Rochester, NY, pp. 22-28.

[29] P.H. Winston [Dec. 1980], Learning and reasoning by analogy, CACM, vol. 23, no. 12, pp. 689-703.

[30] N. Wirth [1973], Systematic programming: An introduction, Prentice-Hall, Englewood Cliffs, NJ.

[31] N. Wirth [1976], Algorithms + Data Structures = Programs, Prentice-Hall, Englewood Cliffs, NJ.

[32] H. Wossner, P. Pepper, H. Partsch, and F.L. Bauer [Summer 1978], Special transformation techniques, Proc. Intl. Summer School on Program Construction, Marktoberdorf, West Germany, pp. 290-321.

[33] L. Yelowitz and A.G. Duncan [Aug. 1977], Abstractions, instantiations and proofs of marking algorithms, Proc. ACM Symp. on Artificial Intelligence and Programming Languages, Rochester, NY, pp. 13-21.

| BIBLIOGRAPHIC DATA SHEET | 1. Report No. UIUCDCS-R-81-1011 | 2. | 3. Recipient's Accession No. |
|---|---|---|---|

4. Title and Subtitle

The Evolution of Programs: Program Abstraction and Instantiation

5. Report Date: June 1981

6.


7. Author(s)

Nachum Dershowitz

8. Performing Organization Rept. No.

9. Performing Organization Name and Address

Digital Computer Laboratory
1304 W. Springfield
Urbana, IL 61801


10. Project/Task/Work Unit No.


11. Contract/Grant No.

MCS-79-04897

12. Sponsoring Organization Name and Address

National Science Foundation
Washington, D.C.


13. Type of Report & Period Covered

14.

15. Supplementary Notes


16. Abstracts

Our goal is to develop techniques for abstracting a given set of programs into a program schema and for instantiating a given schema to satisfy concrete specifications. We suggest the formulation of analogies as a basic tool in program abstraction. An analogy is first sought between the specifications of the given programs. The analogy is then used as a basis for transforming the existing programs into an abstract schema that represents the embedded technique, with the invariant assertions and correctness proofs of the given programs helping to verify and complete the analogy. A given concrete specification of a new problem may then be compared with the abstract specifications of the schema to suggest an instantiation of the schema that yields a correct program. A collection of program schemata is included in an appendix.



17. Key Words and Document Analysis. 17a. Descriptors

abstraction, instantiation, program schemata, analogy, transformations, program manipulation, program development, program correctness

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group


| 18. Availability Statement | 19. Security Class (This Report) UNCLASSIFIED | 21. No. of Pages 63 |
|---|---|---|
| | 20. Security Class (This Page UNCLASSIFIED | 22. Price |