An Approach to Formal Definitions and Proofs of Programming Principles

JAYADEV MISRA, MEMBER, IEEE

Abstract—A method for formal description of programming prinicples is presented in this paper. Programming principles, such as sequential search can be defined and proven even in the absence of an application. We represent a principle as a program scheme which has partially interpreted functions in it. The functions must obey certain input constraints. Use of these ideas in program proving is illustrated with examples.

Index Terms—Program design, program verification, proving program schemas.

I. Introduction

THIS work grew out of an effort to define well-known programming principles, such as sequential search, in a formal fashion. Such principles are common in their usage in programming and teaching about programming. However, no uniform methodology currently exists for defining them. This prevents us from making precise statements about these principles and proving their properties, independent of their implementation and use.

Sequential search, for instance, is commonly used to access every element of a data object. The data object may be an array or a linked list or a binary tree. A program employing sequential search would normally include in it a proof that sequential search is correct. If such a principle can be defined precisely and proven once and for all, then every application of the principle can be proven by showing that input specifications are satisfied by the application, and hence it follows that the output will satisfy the output specification of the principle. Thus, a proof of a principle results in a *proof schema*, which can be used as a basis for proving all applications of the principle.

Formal specifications of principles are important and useful from several considerations. It will be straightforward to find out if the input requirements of a principle are met by a given problem, and hence if the problem could be solved by the application of that principle. The principle can be studied (its properties proved and its performance evaluated) even in the absence of a particular problem. A proof of a principle could be used as the basis of a proof for a family of programs that use this principle. A principle's definition can be the basis for its implementation in any particular instance. In fact, the technique suggested in this paper can be used directly as a programming tool, by appropriately defining the implementation of uninterpreted functions.

Manuscript received August 13, 1976; revised January 12, 1978.

The author is with the Department of Computer Science, University of Texas at Austin, Austin, TX 78712.

It would seem natural that a principle should be defined on abstract data types. However, we use abstract data type in a sense different from Clu [7] or Alphard [10]. A principle is specified to work on any (abstract) data type which has a set of functions defined on it. Data can be accessed/examined/modified using only the given functions. In this sense, our abstract data have much in common with the axiomatic approach for defining data types [7] in that these functions alone define the data type. It may be noted that two pieces of data may be considered equivalent (i.e., as of the same abstract data type) by some principle P_1, though they may be considered different by another principle P_2. This is in contrast with current schemes, where the type of a data object is considered fixed by all procedures. (Hence, objects x, y of identical types are either both legal inputs or both illegal inputs to a procedure.)

Dijkstra [1] has suggested that well-known programming principles, such as linear search, should be isolated and studied independent of their implementations. The method proposed in this paper is a formal way of stating such principles. Notion of "generic procedure" by Gries and Gehani [3] is most relevant to the work reported here. They state, "If we write a procedure to sort an array of values, we are not interested in whether the values are integers or reals or what have you, but instead we are interested in, and our proof of the sort procedure depends on, the fact that the assignment operator: = and the ordering operator $\leqslant$ are defined on the type of array values." They recommend extension to a programming language, such as Pascal, so that a procedure may be called from different program points with actual parameters which are arrays of varying lengths, dimensions, and types of elements. This paper is motivated by similar considerations though the suggested solution is different. Backtrack schema has been studied in considerable detail by Gerhart and Yelowitz [4]. Their work is of the same flavor as the work reported here, though they limit their discussion to backtrack schema only. Program schemas have been studied by a number of researchers; an excellent survey may be found in Manna [8].

A principle is represented as a program schema with partial interpretations of certain functions. The partially interpreted functions must satisfy certain properties called input constraints. Then it is shown that certain propositions (output specifications) hold on termination of the principle. The partially interpreted functions actually define the data object on which the principle operates. We introduce the notion of *proof schema* which forms the basis for proof of a family of "almost identical" procedures.

0098-5589/78/0900-0410$00.75

We describe the basic methodology in Section II. Several examples appear in Section III. Verification issues are considered in Section IV.

II. Basic Methodology

It is best to illustrate the problems and the method with an example. Consider the following program for finding the maximum of an array $A[1 \cdots n]$ of positive integers.

max := $A[1]$;

for i := 2 to n **do if** max < $A[i]$ then **max** := $A[i]$ **end**;

It may be noted that the same program is applicable to an array of reals, to finding the minimum (if we redefine the meaning of "<") and that the same basic idea is applicable to any other data structure where all elements could be accessed one by one through some accessing function. No particular property of integers is being used except that they form a linear ordering with respect to "<". No property of an array is being used except that all its elements can be processed by properly incrementing an index.

These considerations lead us to the following program, which is considerably more general than the previous one; maxfinder (first, last, succ, value, R, max). A description of the parameters, constraints among them, and body of the program follow.

Informal Meaning of Parameters:

first/last refer to first/last element of some structure of elements. (These are actually functions of 0 argument, assuming that they are defined on a fixed structure.)

succ is a successor function, which returns the next element in the structure given a current element, except when the current element is the last element.

value returns the value of an element.

R is a (linear ordering) relation among elements of the structure; i.e., a Boolean-valued function of two arguments.

max is the output; it is of type element of the structure and denotes the largest element with respect to R.

Input Constraints: (Let succ^k, $k \geq 0$ denote k applications of the succ function, let *elem* denote objects of type element.)

1) $\exists k \geq 0$, last = succ^k (first), (starting with first, last will be reached in a finite number of steps through successive application of succ).

2) $R(a,a)$, (R is reflexive). If $a \neq b$ then either $R(a,b)$ or $R(b,a)$.

$R(a,b) \Leftrightarrow \neg R(b,a), \quad a \neq b$ (R is antisymmetric).
$R(a,b) \wedge R(b,c) \Rightarrow R(a,c)$ (R is transitive).

Output Specification: (Max is the maximum)

1) $\forall m, R[\text{value(max)}, \text{value}(a)], a = \text{succ}^m(\text{first}), \quad 0 \leq m \leq k.$

2) $\exists n, R[\text{value}(b), \text{value(max)}], b = \text{succ}^n(\text{first}), \quad 0 \leq n \leq k.$

Body of the principle:

var next: elem;
max := first; next := first;
while next ≠ last **do**
begin next := succ(next);
if R(value(next), value(max)) then max := next **end**;

It is straightforward to show that if the input parameters are of the type as specified in the parameter specification and if the input constraints are met, then the output constraints will be met. The proof uses the following loop invariant.

$\exists p \geq 0$, next = succ^p(first), and
$\forall q \leq p, R$(value(max), value(succ^q(first))), and
$\exists r \leq p$, max = succ^r(first)

Termination of the program can be shown by a direct application of input constraint 1). The loop invariant along with the loop termination condition implies the output specification.

A principle may be invoked by "calling" it with proper input parameters. In order to use max-finder to find the maximum of an array $A[1 \cdots n]$ of integers, we may use the following calling sequence.

(Element is an index);
function first ≡ 1;
function last ≡ n;
function succ(i) ≡ i + 1;
function value (i) ≡ $A[i]$
function R (a,b) ≡ $a > b$; a,b integers

In order to show that max = maximum ($A[1] \cdots A[n]$) on termination of max-finder, we have to show that the input constraints of max-finder are met and that output specifications of max-finder suitably interpreted imply the desired ouput condition. Thus, we need to show that

i) starting with i = 1 and successively updating i to i + 1, we will eventually reach n.

ii) ">" is a linear ordering among integers. Furthermore, in order to guarantee that all elements are looked at, we need to show that

iii) all indices in $1 \cdots n$ are obtained by starting with i = 1 and successively updating i by 1.

Max-finder may be used to find the extremal element with respect to any linear ordering in any data structure whose elements can be accessed through a suitable traversal function.

Formal Definition of a Principle

1) A principle is defined to operate on one or more abstract data types and to return results of some specific type.

2) Abstract data types are defined axiomatically by a set of functions along with certain constraints on them. These are termed the "input constraints." The functions are uninterpreted entities inside a principle's body. They are solely used to access/examine/modify the abstract data objects.

3) Abstract data objects cannot be accessed/examined/modified by any other means by the principle.

4) Two data objects are considered to be of identical type by a principle if they have the required functions defined on them. Thus, maxfinder considers a tree (of character strings), a linked list (of reals) and an array (of integers) to be identical, assuming that suitable traversal routines and orderings (R) have been defined for each one of them. Some other principle may differentiate between linked lists and arrays, for instance.

Current methods for abstract type definition in programming languages take the view that procedures should be de-

fined to work on fixed types. We have proposed that certain (procedures) principles should be defined on all types satisfying certain axioms. This method can be used in conjunction with usual techniques for abstract data type definition.

5) Output parameters are objects of specified types.

6) The only operation that is defined for all types is assignment and equality comparison. Hence, any two objects of identical types may be assigned to each other or compared.

7) An invocation of a principle is an interpretation for each input parameter (functions) and a proof that input constraints are satisfied. It then follows that the corresponding output constraint suitably interpreted holds on termination of the principle. Thus, we have *factored* the proof of some procedure into a *proof schema* and a proof that a certain interpretation satisfied the (input constraints) axioms of the schema, and a proof that the output constraints of the proof schema along with the given interpretation imply the desired output condition of the procedure. Clearly, the same proof schema is applicable as a basis for proofs of a family of different procedures.

There is a great deal of similarity between principles and procedures. Both techniques are means of specification of general computation rather than computational steps of any particular instance. Both achieve their power through parameterization; that the same computation may be applied to objects belonging to certain well-defined classes. Both techniques seem crucial for transparent programming since they implement an abstraction in a general form.

The major difference between the two techniques seems to be the abstract data type definition. Principles, in this sense, may be regarded as a generalization of procedures. Each uninterpreted function, suitably interpreted usually results in a procedure. As a procedure gives rise to many possible invocations, a principle may be the basis for many possible procedures. As a proof of a procedure could be applied in proving effects of certain instances of the procedure call, a proof of principle may serve as a basis for proofs of all instances of the applications of the principle.

III. Examples

Two examples are given to illustrate the ideas of the previous section.

Example 1 (Sequential Search)—It is required to search over every element of a data object using a function (coroutine) *traverse*. Program is terminated as soon as an element meets a *criterion* or traversal is *complete*. A Boolean variable *success* is set to true if and only if criterion is ever met.

```
var v: elem;
success := false; traverse (v);
while not success and not complete do
  if criterion (v) then success := true else traverse (v).
```

Description of Functions:

traverse: It is a coroutine that returns new element or sets a Boolean flag *complete* to true if there is no more element.

criterion: It is a Boolean-valued function that returns true if the given element meets the requirement.

Input Constraints: None.

Output Specification:

1) Program terminates.

2) Let $T(v)$ be a predicate which denotes that an element v will be obtained if *traverse* is applied a sufficient number of times.

$$[\exists v, T(v) \wedge \text{criterion}(v)] \Leftrightarrow [\text{success} = \text{true}].$$

Example 2 (Gradient Search)—The following program finds an optimal *solution* using gradient search. There are a finite number of solutions to some problem. Each solution has one or more *neighbors*. Each solution has a positive integer *value*. If a solution is not optimal (does not have the maximum value) then there is a neighboring solution with a higher value. Gradient search starts with an arbitrary solution. If a current solution is not optimal, it moves to the neighboring solution of highest value.

```
var current-soln, new-soln, t : solution; max value : integer;
  choose (new-soln);
  repeat
  current-soln := new-soln; max-value := 0;
  for t ∈neighbor (current-soln) do
    if value (t) > max-value then
    begin
    new-soln := t;
    max-value := value(t)
    end
  until value (new-soln) ≤ value (current-soln);
```

Description of Functions:

choose: returns (nondeterministically) any solution.

neighbor: for any solution, returns the *set* of all neighboring solutions.

value: returns the value of any solution as a positive integer.

Input Constraints:

1) There are a finite number of solutions.

2) value$(t) \geq 0$, for every solution t.

3) Every nonoptimal solution has a neighbor having a higher value. Clearly, an optimal solution is one which does not have a neighbor having a higher value. Define a predicate *optimal*, optimal$(S) \Leftrightarrow \forall\, t[\text{value}(S) \geq \text{value}(t)]$. We require that,

$$\neg\text{optimal}(S) \Rightarrow \exists t \in \text{neighbor}(S)\ [\text{value}(S) < \text{value}(t)].$$

Hence, optimal$(S) \Leftrightarrow \forall t \in \text{neighbor}(S)\ [\text{value}(S) \geq \text{value}(t)]$.

Output Specification:

1) Program terminates.

2) Optimal (current-solution).

IV. Verification Issues

A principle may be verified with respect to the input constraints and output specification in a manner similar to the usual program verification [5]. We are restricted to using the axioms involving functions on the abstract data type and the assignment axiom and usual rules of inference for the control structures used. We illustrate the method with verification of Example 2, Gradient Search.

We first show how an auxiliary fact may be proven from input constaints. Define a predicate *reachable*, where reachable (u,s) denotes that a solution u can be reached from solu-

tion s by moving through some sequence of neighbors. More formally,

$$\text{reachable}(u,s) \equiv u = s \text{ or } u \in \text{neighbor}(s) \text{ or } [u \in \text{neighbor}(t) \text{ and reachable } (t,s)].$$

We wish to show that the input constraints imply that starting with any initial solution, it is possible to reach an optimal solution.

Proposition

$$\forall S,\ \exists u\ [\text{reachable}(u,s) \text{ and optimal}(u)].$$

Proof: For any initial solution S_o, let U_o be such that

1) reachable (U_o, S_o),

2) value(U_o) is maximum among all reachable solutions from S_o.

Since there are a finite number of solutions (input constraint 1), U_o is well defined. If U_o is not optimal, according to input constraint 3), it has a neighbor U, having higher value. But then U satisfies Conditions 1) and 2) above. Contradiction! □

Proof of this fact is important in order to make sure that a program can be written meeting the output specifications. We now give an informal proof of termination.

Proposition

value(new-soln) is (strictly) monotone increasing in each iteration.

Proof: Follows from the loop exit condition. □

Termination follows since there are a finite number of solutions and value(new-soln) strictly increases in each iteration.

Proposition

At the termination of the loop, optimal (current-soln).

Proof: It may be shown (easily) that the following proposition P holds after every execution of the loop body.

$P \equiv$ value(new-soln) $\geq$ value(t)), $t \in$ neighbor(current-soln) **and** new-soln ϵ neighbor (current-soln).

On termination of the loop, we may assert P and value(new-soln) $\leq$ value (current-soln). This is equivalent to,

$\forall t$, $t \in$ neighbor (current-soln), value(current-soln) $\geq$ value(t).

Using input constraint 3), it follows that

optimal (current-soln). □

It may be noted from this proof that a level of abstraction has been introduced by the definition of the principle. Neither the representation nor any other facts about the abstract data object *solution* are used, except those explicitly stated in the input constraint. The proof given is actually a *proof schema* in that every problem using gradient search may be proven in a similar manner by suitably defining the functions, choose, neighbor, value.

V. Summary and Conclusion

We have described a technique for formally defining and proving programming principles independent of their application. This usually leads to more transparent programming and more structured proofs. We particularly note the following advantages.

1) Formal specification of programming principles: A programming principle can be stated and proved formally in the absence of a problem. Conditions for applicability of a principle are formally stated.

2) Proof factorization: The proof of the principle serves as a basis for any application of the principle. In particular, it is sufficient to show that the input specifications are met, in order to prove that the corresponding output specifications hold on termination.

3) The definition of the principle can be used directly as a programming tool, much like a procedure in the current programming languages. Unlike a procedure, the inputs are now functions (rather than data objects). Burden of proving that the principle is being called correctly, i.e., the input specifications are being met, is up to the programmer.

4) Use of principles leads to a new form of abstraction and to more transparent programs and proofs. Aspects that are problem dependent are separated from aspects that may be considered inherent in the algorithm.

We believe that programming languages currently in use can be easily extended to facilitate the definition of principles. A language such as Pascal allows the passing of functions and procedures as parameters. Furthermore, procedures for assignment and for checking the equality of two objects of some abstract data type should be defined. These simple extensions are sufficient to describe most principles.

It may often be necessary to optimize the invocation of a principle. For instance, the general gradient search schema could be considerably simplified and made efficient for a particular case, such as linear programming problems. These optimizations should best be performed manually along with a proof that the optimizing steps preserve the input, output relationships.

References

[1] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming*, C. A. R. Hoare, Ed. New York: Academic, 1972.

[2] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Applied Mathematics, American Mathematical Society*, J. T. Schwartz, Ed., Providence, 1967.

[3] D. Gries and N. Gehani, "Some ideas on data types in high level languages," *Commun. Ass. Comput. Mach.*, vol. 20, no. 6, pp. 414–420, June 1977.

[4] S. L. Gerhart and L. Yelowitz, "Control structure abstractions of the backtracking programming technique," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 285–292, Dec. 1976.

[5] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, no. 10, pp. 576–583, 1969.

[6] C. A. R. Hoare and N. Wirth, "An axiomatic definition of the programming language PASCAL," *Acta Informatica*, vol. 2, pp. 335–355;

[7] B. H. Liskov and S. N. Zilles, "Specification techniques for data abstraction," *IEEE Trans. Software Eng.*, vol. SE-1, Jan. 1975.

[8] Z. Manna, *Mathematical Theory of Computation*. New York: McGraw-Hill, 1974.

[9] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 221–227, 1971.

[10] W. A. Wulf, R. L. London, and M. Shaw, "Abstraction and verification in Alphard," Information Sciences Institute, Univ. Southern Calif., ISI/RR-76-46, June 1976.

Jayadev Misra (S'71–M'72), for a photograph and biography, see p. 69 of the January 1978 issue of this TRANSACTIONS.