Deriving programs using generic algorithms

by V. R. Yakhnis
J. A. Farrell
S. S. Shultz

We suggest a new approach to the derivation of programs from their specifications. The formal derivation and proof of programs as is practiced today is a very powerful tool for the development of high-quality software. However, its application by the software development community has been slowed by the amount of mathematical expertise needed to apply these formal methods to complex projects and by the lack of reuse within the framework of program derivation. To address these problems, we have developed an approach to formal derivation that employs the new concept of generic algorithms. A generic algorithm is one that has (1) a formal specification, (2) a proof that it satisfies this specification, and (3) generic identifiers representing types and operations. It may have embedded program specifications or pseudocode instructions describing the next steps in the stepwise refinement process. Using generic algorithms, most software developers need to know only how to pick and adapt them, rather than perform more technically challenging tasks such as finding loop invariants and deriving loop programs. The adaptation consists of replacing the generic identifiers by concrete types and operations. Since each generic algorithm can be used in the derivation of many different programs, this new methodology provides the developer with a form of reuse of program derivation techniques, correctness proofs, and formal specifications.

The use of formal software development methods, such as formal specification, program derivation, and proofs of correctness of algorithms, has been advocated by the academic computer science and software engineering community for about two decades. Yet, even though this mathematical approach has enormous potential for producing very high-quality software, its use so far within the software development community has not been commensurate with its potential. Although many programmers agree that the production of programs that are mathematically proved to be correct is a desirable goal, it appears that the mathematical skill level required of the program development team to produce correctness proofs may be too high for their widespread acceptance.

The original formal approach, called program verification, was to develop an algorithm by traditional means and then create its correctness proof, thereby revealing any faults in the algorithm. It soon became apparent that creating the proof independently of creating the algorithm made the proof step much too complex, since the formal logic structure of the algorithm had to be extracted before the proof could be formulated. To counter this problem, several researchers (see Dijkstra,[1] Gries,[2] Kaldewaij,[3] and Cohen[4]) proposed a technique in which the algorithm and the proof are developed together, with elements of the proof actually preceding the code. In this approach, both the algorithm and its proof are derived from its formal specification.

Although the program derivation approach addressed many difficulties in producing mathematically correct software, the mathematical skill level required was not appreciably reduced.

Our methodology is applicable to a vast segment of the program development process.

Therefore, when the IBM Glendale Programming Laboratory in Endicott, New York, began investigating the use of formal methods to help improve the quality of our commercial software products, we looked for ways to lower the level of mathematical skill required. Our approach was to "hide" most of the mathematical techniques involved in program derivation in a number of prederived program templates that we call *generic algorithms*.

A generic algorithm has the following features:

- It has a formal specification.
- It is formally proved to satisfy this specification.
- It may have one or more generic identifiers representing data types or operations.
- It may have embedded program specifications or pseudocode instructions describing the next steps in the stepwise refinement process.

Given a collection of generic algorithms, the suggested process of developing a program from a formal specification could be informally described as follows. Instead of directly using the program derivation techniques, select a suitable generic algorithm from our collection and adapt it by replacing its generic identifiers by conceptually similar data types and operations. The selection is based either on comparison between the logical structure of the given formal specification and the specifications of the generic algorithms, or intuitive comparison of the desired "behavior" of the algorithm being designed with the "behaviors" of the generic algorithms, or both. Then check the correctness of certain logical assertions linking the original formal specification and the specification of the generic algorithm modified by the aforementioned replacement of the data types and operations.

These "linking" assertions are discussed later in the paper. Their correctness ensures that the adapted generic algorithm satisfies the original formal specification. We call this methodology "deriving programs using generic algorithms." Just as with traditional program derivation techniques, our methodology is applicable to a vast segment of the program development process, from high-level design to coding.

Within the framework of our approach, generic algorithms serve as reusable, transformable building blocks in the formal derivation of programs. They extend the paradigm of program reuse to the reuse of program correctness proofs, formal specifications, and design steps. Generic algorithms alleviate the problem of mathematical skill levels by freeing the programmer from the search for loop invariants and from providing respective interim proofs. Thus they enable programmers to derive mathematically correct programs from formal specifications with only an occasional need to consult experts in program derivation and program correctness proofs.

Our experience includes the application of formal derivation with generic algorithms in the Conversational Monitor System (CMS) component of the Virtual Machine/Enterprise Systems Architecture* (VM/ESA*) operating system.[5]

Prior to introducing generic algorithms, we describe the basic concepts of formal specification in the next section. This is done not only to provide the necessary background, but also to familiarize programmers with our style of writing formal specifications, which is tailored to the use of generic algorithms. After these preliminaries, we describe the notion of generic algorithms and their use in the succeeding section.

The main body of this paper is intended to provide a description of our methodology rigorous enough to be both convincing and understandable to program developers. The appendices contain mathematical details necessary for proving the validity of our approach. We hope that both program developers and computer scientists interested in program derivation will find this work useful.

Here are some of our notation conventions:

- Algorithms are represented by cursive capital letters like $\mathcal{F}$ or $\mathcal{G}$.
- Boolean expressions are represented by Greek letters like φ, ψ, or γ.
- Other expressions are represented by capital letters in italic, like E.
- Logical assertions are represented by either Greek letters or capital letters in italic.
- Sets are represented by capital letters with double lines, like $\mathbb{X}$, $\mathbb{Y}$.

Specifications and algorithms

Informal, rigorous, and formal specifications. A specification for an algorithm is a statement describing its behavior during execution. Usually this description is limited to stating under what conditions the algorithm may begin its execution and also what kind of results are expected after the algorithm terminates. However, a specification should avoid spelling out how to reach those results, leaving the "how" open to various implementations. Essentially the same specification may be represented in numerous formats, which in turn may be separated into three broad categories: informal, rigorous (sometimes called informal rigorous[6] or semiformal), and formal.

Informal specifications use natural languages (English in our case) with only occasional usage of mathematical symbols. Rigorous specifications use more or less informal mathematical notation with the usual mathematical conventions applied. They are used, for example, by the Dijkstra-Gries school of program derivation.[1-4,7] Finally, formal specifications require the use of a formal specification language, e.g., Z,[8-11] VDM,[12] and various algebraic specification languages such as CLEAR,[13] Larch,[14] and many others.

Since our purpose is to create very high-quality software, every time we create an algorithm $\mathcal{F}$ we must ensure that it satisfies the specification. The only reliable way to ensure satisfaction is to create a correctness proof. Since such correctness proofs require having either a rigorous or a formal specification, our approach works best in the frameworks of either rigorous or formal specifications. However, the generic algorithms approach is so flexible that it also permits the users to create quality software without strictly confining them to either rigorous or formal specifications. Indeed, since rigorous or formal specifications and correctness proofs are implicitly included in generic algorithms, using them as software building blocks will increase the software quality even if the overall specification is informal.

A specification for an algorithm is a statement describing its behavior during execution.

In this paper we use rigorous specifications in the Dijkstra-Gries manner. We also provide several enhancements aimed at accommodating generic algorithms. Since the existing mathematical conventions sometimes do not provide standards for certain elements of notation and since Z provides such standards for most notations needed in program derivation, we, similar to Morgan,[15] often use conventions from Z in addition to those used by the Dijkstra-Gries school.

Finally, we feel that such languages as CLEAR or Larch may significantly enhance the effectiveness of our approach, provided one has easy-to-use tools for checking the syntax of specifications and verifying algorithm correctness assertions. We quote from Guttag and Horning: "Are formal specifications too mathematical to be used by typical programmers? No. Anyone who can learn to read and write programs can learn to read and write formal specifications. After all, each programming language is a formal language."[14]

A style of writing specifications tailored to using generic algorithms. In the following subsections we discuss the development of our approach.

The precondition, the postcondition, and the constraint. Most of the approaches (whether informal, rigorous, or formal) describe specifications in two layers. The first layer defines the *algorithm states* in terms of *identifiers*[16] and *types*, where identifiers are placeholders for values and types are "containers" having those values (usually with some attached operations acting upon the

values). We define an algorithm state as a map from the identifiers into the values contained in the respective types. In regard to its states, we think of an algorithm as a machine transforming these states from one into another.

The second layer defines the desired algorithm behavior in terms of a pair of properties of the algorithm identifiers, say P and Q, where:

- The property P is called the *precondition*.
- The property Q is called the *postcondition*.

The precondition describes the possible values of the identifiers necessary to initiate the algorithm, run it successfully, and terminate. The postcondition describes the allowable values of the algorithm identifiers after the algorithm successfully terminates. In other words, both the precondition and the postcondition describe subsets of the collection of all algorithm states. Using the "state machine" metaphor, we can think of the behavior of an algorithm defined by a specification as a finite sequence of state transitions that begins in the subset of algorithm states described by the precondition and ends in the subset of algorithm states described by the postcondition.

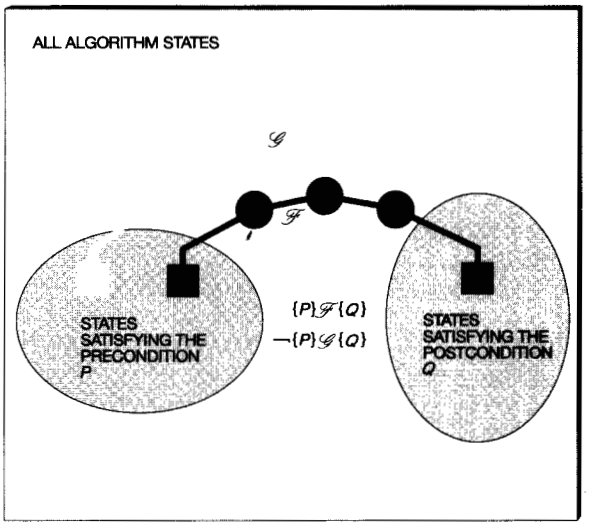
In the Dijkstra-Gries notation the assertion "an algorithm $\mathcal{F}$ satisfies a specification with the precondition P and the postcondition Q" is written as $\{P\}\ \mathcal{F}\ \{Q\}$. In other words, $\{P\}\ \mathcal{F}\ \{Q\}$ holds if for every state satisfying the property P:

- The algorithm $\mathcal{F}$ initiates, runs successfully, and terminates.
- After the termination of $\mathcal{F}$ the resulting state (also called the final state) satisfies the property Q.

In Figure 1 the darker trajectory depicts the behavior of an algorithm $\mathcal{F}$ with $\{P\}\ \mathcal{F}\ \{Q\}$ and the lighter trajectory depicts the behavior of an algorithm $\mathcal{G}$ with $\neg\{P\}\ \mathcal{G}\ \{Q\}$ (recall that "$\neg$" stands for "not").

Although intuitively simple, such treatment does not provide any means to restrict the behavior of an algorithm in between the initiation and termination. We remedy this situation by adding to the precondition and the postcondition a third property of the algorithm states called the *constraint* with the following meaning. We say that an algorithm $\mathcal{F}$ satisfies a specification with the precondition P, postcondition Q, and a constraint C (and write it as $\{P\}\ \mathcal{F}/C\ \{Q\}$) if for every state satisfying both the properties P and C:

Figure 1 An algorithm behavior defined by a precondition and a postcondition

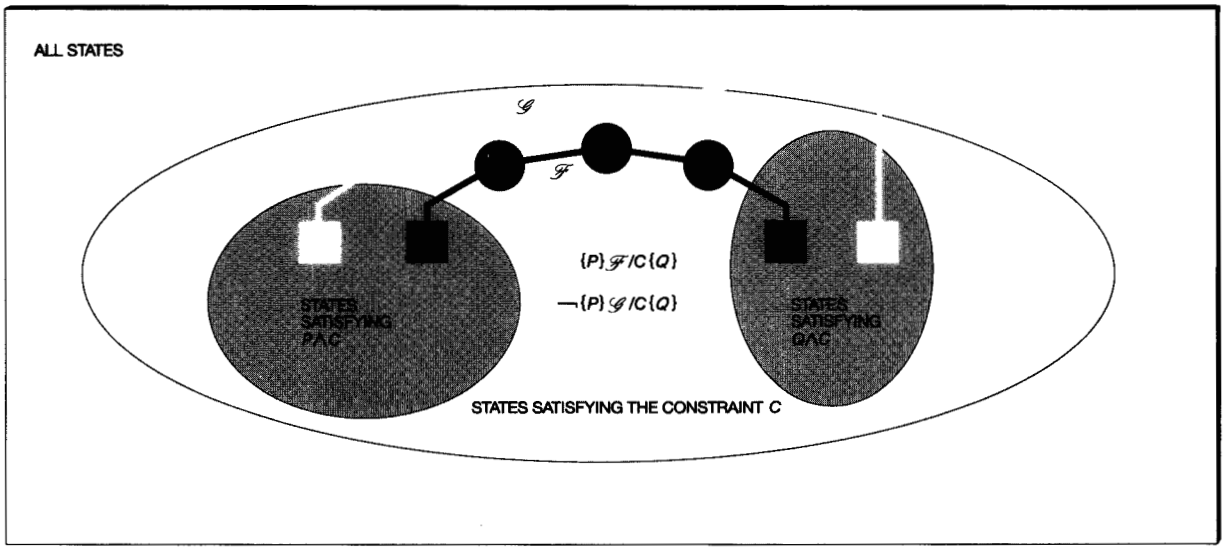


- The algorithm $\mathcal{F}$ initiates, runs successfully, and terminates.
- After the termination of $\mathcal{F}$ the final state satisfies both the properties Q and C.
- All the states reached by $\mathcal{F}$ between the initial state and the final state satisfy the property C.

Recalling that "$\wedge$" stands for "and," it is easy to notice that $\{P\}\ \mathcal{F}/C\ \{Q\}$ implies $\{P \wedge C\}\ \mathcal{G}\ \{Q \wedge C\}$, whereas the opposite may not be true. In Figure 2 the darker trajectory depicts the behavior of an algorithm $\mathcal{F}$ with $\{P\}\ \mathcal{F}/C\ \{Q\}$, and the lighter trajectory depicts the behavior of an algorithm $\mathcal{G}$ with $\neg\{P\}\ \mathcal{G}/C\ \{Q\}$. Note, however, that for the algorithm $\mathcal{G}$, $\{P \wedge C\}\ \mathcal{G}\ \{Q \wedge C\}$ is satisfied.

Some approaches (Dijkstra-Gries, VDM) do not have a direct analog of the notion of constraint. The closest analog they employ is the notion of *invariant*, i.e., a property that must be satisfied by both the initial and the final states but, unlike constraints, may be violated in between. Although we find invariants useful, they cannot serve the same purpose as constraints. Other approaches (like Z or algebraic specification languages) have analogs of constraints. In Z the notion has the

Figure 2 An algorithm behavior defined by a precondition, a postcondition, and a constraint



same name (hence our name), whereas in algebraic specification languages the notion can be represented as a collection of axioms. However, since these approaches concentrate on specification and verification issues and not on program derivation, our exploiting of constraints within the framework of sequential approaches to program derivation is new. Also, since we have not seen in the literature a convenient collection of proof rules covering the notion of constraint, we give our own rules in Appendix B.

We deem the notion of constraint to be convenient for the following reasons:

1. It may help to extend sequential algorithm development approaches to a concurrent setting. Indeed, often all of the properties needed to prevent a sequential algorithm from unduly influencing other concurrently running processes can be incorporated in the constraint. (For example, the Owicki and Gries notion of "interference freedom"[17] can be represented via constraints.) Then a sequential development approach using constraints can be employed to create this algorithm. We have done this while applying our methodology to the multitasking component of CMS.[5]
2. The properties of constant identifiers (i.e., those that the algorithm is not allowed to change) are the same throughout the execution. Therefore, it is more reasonable to include them in the constraint rather than (as usual) in the precondition.
3. Since the constraint is always true, the assertions included in it are convenient to use throughout the algorithm structure in the local correctness proofs.

The first specification layer: Defining the data structure. A statement establishing an association between an identifier and a data type is usually called a *declaration*. A combination of several declarations and a constraint is called a *data structure*. The first layer of a specification describes the data structure for the algorithm being specified. This data structure completely describes the space of states upon which the algorithm is allowed to operate. We now give some technical details of how to describe the data structures.

First note that some identifiers are standard, e.g., the symbol for addition "+" or the symbol for the set of nonnegative integers "$\mathbb{N}$", and thus they may be used but need not be defined in the data structure. We divide all the identifiers into *object identifiers* and *type identifiers*.[18] Type identifiers

Figure 3 Data structure from the specification of an algorithm computing the factorial function

[Factorial_Data_Structure
/* ! is a function that takes a nonnegative integer value and creates a positive integer value. The function ! may not occur in the algorithm. */

spec con ! : $\mathbb{N} \rightarrow \mathbb{N}$	/* $\mathbb{Z}$ = all integers, $\mathbb{N}$ = all nonnegative integers */
con *nzero* : $\mathbb{Z}$	/* input value */
var *r* : $\mathbb{Z}$	/* output value */

constraint
/* Recall that "$\wedge$" means "and," "A $\Rightarrow$ B" means "if A holds then B also holds," and that "$(\forall n : \mathbb{N} \cdot \ldots)$" means "for all n from the set $\mathbb{N}$ the following holds . . . ." Identifiers with the scope limited to a formula (like n here) are called *bound variables*. They are distinct from object identifiers. */

$nzero \geq 0 \wedge (\forall\, n : \mathbb{N} \cdot (n = 0 \Rightarrow n! = 1) \wedge (n \neq 0 \Rightarrow n! = (n - 1)! * n))$
]

are used to denote collections of data objects, whereas object identifiers are used to denote individual data objects. Suppose that x, f, and g are object identifiers and $\mathbb{X}$ and $\mathbb{Y}$ are type identifiers. We use the following declarations:

- x:$\mathbb{X}$ /* any value associated with name x must be taken from the set $\mathbb{X}$ */
- f:$\mathbb{X} \rightarrow \mathbb{Y}$ /* f is a function taking values from $\mathbb{X}$ and returning values from $\mathbb{Y}$ */
- g:$\mathbb{X} \nrightarrow \mathbb{Y}$ /* g is a partial function taking values from $\mathbb{X}$ and returning values from $\mathbb{Y}$ */

Functions with more than one variable are represented using the Cartesian product "$\times$" e.g., "f:$\mathbb{X}\times\mathbb{Y}\times\mathbb{W}\rightarrow\mathbb{U}$" means that the first argument is taken from $\mathbb{X}$, the second from $\mathbb{Y}$, and the third from $\mathbb{W}$. The construction describing the types of the arguments and the type of the returned value for a function is called the signature of the function. For instance, the signature of f is $\mathbb{X}\times\mathbb{Y}\times\mathbb{W}\rightarrow\mathbb{U}$. The same is true for partial functions, e.g., g:$\mathbb{X}\times\mathbb{Y}\times\mathbb{W}\nrightarrow\mathbb{U}$.

We separate object identifiers into *algorithm identifiers* and *specification identifiers*. The former can be used either in specification, pseudocode instructions, or in standard instructions of the algorithm being specified. The latter can be used only in specification or pseudocode instructions of the algorithm being specified. We discuss pseudocode instructions and standard instructions later. As usual, the algorithm identifiers are separated into *constant*s (designated as "**con**") and *variable*s (designated as "**var**"). The former may not be changed by the algorithm, but the latter are allowed to be changed. This separation usually amounts to excluding constants from the left-hand parts of assignments while not placing such restrictions on the variables.

Some of the variables may be marked as "input," which means that they are considered to possess a value in the initial state of any computation on the respective data structure. The variables that are not so marked are considered to be uninitialized, i.e., any algorithm trying to evaluate an expression with noninput variables before some values are explicitly assigned to them will be forced to abort.

We separate specification identifiers into *specification constant*s (designated as "**spec con**") and *specification variables* (designated as "**spec var**"). The former may not be changed by the pseudocode instructions, but the latter can be changed. The above notions (excepting specification variables) are illustrated in the data structure from the specification of an algorithm computing the factorial function, shown in Figure 3. The usage of specification variables will be illustrated in the next section.

Note that variable r is not marked as input, and thus it is uninitialized. In this example, the factorial function "!" is declared as a specification constant since we may not use it as a function inside the program. However, we need this symbol in order to state our intentions (the postcondition $r = nzero!$, see the full specification in the subsection that discusses the second specification

layer). By declaring *nzero* as a constant, we forbid changing *nzero* by any algorithm satisfying the specification. Finally, the constraint is essentially a definition of the factorial function.

In most of the literature, for example, in Kaldewaij[3] and Cohen,[4] specification constants are not explicitly declared. In Morgan,[15] specification constants are called logical constants, and they are explicitly declared, whereas what we call constants are not explicitly declared. By explicitly declaring all variables and keeping them classified by purpose, we have the advantage of clearly seeing what usage is legal for each variable. Finally, our notion of specification variables is new. We illustrate their usage in the next section.

Here are some additional notation conventions:

- Specification identifiers are represented by words in nonitalic capital letters like "BOUND" or special symbols like "!".
- Algorithm identifiers are represented by words in lowercase italic, like "*nzero*".
- Bound variables are represented by nonitalic letters in lower case, like n, x, y.

Constructions. Constructions are convenient means to abbreviate certain lengthy descriptions. We illustrate this notion by taking examples from the specification for Star_Recursion in the following section. We distinguish three kinds of constructions:

- Type constructions—Single type identifiers are the simplest type constructions. The rest are the result of building sets by the usual means. For example, $\mathbb{X}\times\mathbb{X}\rightarrow\mathbb{X}$ is the set of all possible binary operations on $\mathbb{X}$. $\{y{:}\mathbb{Y}|low\_bound(y)\}$ is the set of all such y from the set $\mathbb{Y}$ that *low_bound*(y) is true. $\mathbb{Y}\rightarrow\mathbb{B}$ is the set of all Boolean-valued functions taking values from $\mathbb{Y}$.
- Expressions (also called terms)—They are the object identifiers together with various combinations of applications of functions to the object identifiers. For example, y, BOUND($tail(y)$), G($tail(y)$) $\star$ $head(y)$, $base(y)$, etc.
- Function constructions—They are a useful technical tool for describing maps translating one data structure into another data structure, though they are not crucial for understanding such maps. We describe them in Appendix A.

As a shorthand we permit using the set constructions in the declarations in lieu of type identifiers. For example, suppose that we would like to introduce a constant c from the set $\mathbb{Y}$ satisfying the property *low_bound*(c) = **true**. The standard way to do so would be to add $c{:}\mathbb{Y}$ to the declarations and to add *low_bound*(c) to the constraint (recall that in logical assertions *low_bound*(c) stands for *low_bound*(c) = **true**). However, the above permission means that we could just add $c{:}\{y{:}\mathbb{Y}|low\_bound(y)\}$ to the declarations without explicitly adding anything to the constraint. Implicitly, however, *low_bound*(c) = **true** would still be part of the constraint.

Generics. A crucial part of our algorithm development methodology is the usage of generics, i.e., generic data types and data objects. Generics are not new. Programming languages such as Ada** and approaches such as Z or algebraic specification languages have used them for quite a while. However, we feel that our approach utilizes the generics more fully because our usage of generics permits us to reuse the proofs and our specification variables constitute a new type of generics.

A *generic type identifier* does not correspond to any concrete set but has attached to it a collection of axioms describing properties of its abstract elements. For a given specification, we list all of the required generic type identifiers (if any) in the section designated "**gen**" in the beginning of the specification (e.g., **gen** $\mathbb{X}$, $\mathbb{Y}$), and we add all their axioms (if any) to the constraint.

We define *generic object identifiers* as those object identifiers whose type is a type expression containing a generic type identifier. In addition, we call a construction (see Appendix A) generic if it contains an occurrence of a generic identifier.

Our experience has convinced us that, while either creating the generic algorithms or using them to derive other algorithms, it would be very convenient to have at the tip of one's fingers an extensive collection of standard generic type constructions (also called abstract data types), such as linked lists, bags, sets, etc. An excellent such collection is in Guttag and Horning.[14]

The second specification layer: Defining the invariant, the precondition, and the postcondition. Given a data structure, we describe the intended algorithm behavior by means of the invariant, the precondition, and the postcondition in terms of

Figure 4 A formal specification for an algorithm computing factorials

```
[Factorial_Spec
/* The purpose of an algorithm satisfying Factorial_Spec is to compute nzero! for nonnegative nzero and put the result
into r. Recall that the "factorial" function "!" could be defined as follows. For a positive integer n, n! = 1*2* . . . *n.
Also, 0! = 1. The factorial satisfies the following "star recursion" property: for a positive integer n, n! = (n − 1)! * n. */

Factorial_Data_Structure                    /* see earlier subsection on the first specification layer */

postcondition r = nzero!]
```

the identifiers from the data structure. Having discussed the latter two at length, we repeat the definition of the former: the invariant is a property that must be satisfied by both the initial and the final states but may be violated in between. We illustrate these notions by giving a formal specification for an algorithm computing factorials as presented in Figure 4.

Note that we consider the constraint, the invariant, and the precondition sections of a specification to be optional, and we may omit any of them if the respective logical assertion is trivially true. Finally, following Wirsing,[19] we call a specification *sensible* if its data structure has at least one state satisfying both the invariant and the precondition. One should not waste time by working with a nonsensible specification.

On the basis of the specification concepts we have introduced, we summarize what it means for an algorithm $\mathcal{F}$ to satisfy a specification *SPEC* written in our style. Let SC, SV, C, and V be, respectively, the collections of specification constants, specification variables, constants, and variables from *SPEC*, and let *Constr*, *Inv*, *Pre*, and *Post* be, respectively, the constraint, the invariant, the precondition, and the postcondition from *SPEC*. We say that $\mathcal{F}$ satisfies *SPEC* if for every initial state satisfying $Constr \wedge Inv \wedge Pre$, the following is true:

1. The algorithm $\mathcal{F}$ successfully terminates.
2. The final state satisfies $Constr \wedge Inv \wedge Post$.
3. During the execution, *Constr* is always **true**.
4. The variables from SC and SV do not occur in the algorithm $\mathcal{F}$.
5. During the execution the value of each variable from C is never changed.
6. During the execution, the values of all the variables always belong to their respective types.

We denote the first three conditions as $\{Inv \wedge Pre\}$ $\mathcal{F}/Constr$ $\{Inv \wedge Post\}$. The Dijkstra-Gries notation for the first two conditions is $\{Constr \wedge Inv \wedge Pre\}$ $\mathcal{F}$ $\{Constr \wedge Inv \wedge Post\}$.

Algorithms and programs. We now describe aspects of the algorithms and programs involved in our approach.

Building algorithms from standard instructions and pseudocode instructions. Given a specification, our intuitive concept of an algorithm satisfying it is a state machine transforming the states defined by the data structure of the specification. However, besides the variables declared in the data structure of the specification, the algorithm may have some additional variables serving an auxiliary purpose in the sense that there is either no input or output associated with them (e.g., array indices, loop counters, etc.). We call them *work variables*, and we designate their scope by the additional algorithm brackets "|[" and "]|". (As an example, see Star_Recursion in the next section.) Finally, we call the union of the data structure of the specification and the declarations of the work variables the *algorithm data structure*.

We now describe how the algorithms transform the states defined by the algorithm data structure. Since a formal treatment (see Harel,[20] and Loeckx and Sieber[21]) is beyond the scope of this paper, we give an intuitive description of how the algorithms are built from the atomic parts and how these atomic parts work on an imaginary computer. It will conceptualize the notion of "execution" for the algorithms, even if, in general, the algorithms are not executable by any real computer.

We think of an algorithm as a combination of standard instructions and pseudocode instructions. We describe the former by using the concepts developed by Edsger Dijkstra in the 1970s and further developed in the 1980s by David Gries and

Table 1 Subset of Dijkstra-Gries Instructions

Instructions in Gries-Dijkstra Notation	Behavior During Execution
Skip *skip*	*Step 1.* Do nothing; *Step 2.* Terminate.
Composition /* $\mathcal{F}$ and $\mathcal{G}$ are algorithms. */ $\mathcal{F}$, $\mathcal{G}$	*Step 1.* Execute $\mathcal{F}$, *Step 2.* Execute $\mathcal{G}$, *Step 3.* Terminate.
Assignment /* x is a variable and E is an expression of the same type. */ x: = E	*Step 1.* Compute the value of the expression E in the current program state. If E is undefined then crash. Otherwise go to the next step; *Step 2.* Get the new program state by replacing the value of the variable x by the value of E and leaving the values of all other variables unchanged; *Step 3.* Terminate.
Simple IF /* γ is a Boolean expression and $\mathcal{F}$ and $\mathcal{G}$ are algorithms. */ **if** $\gamma \rightarrow \mathcal{F}$ $\square \neg\gamma \rightarrow \mathcal{G}$ **fi**	*Step 1.* Evaluate the guard γ. If it is undefined then crash. Otherwise go to the next step; *Step 2.* If γ evaluates as **true,** execute $\mathcal{F}$. Otherwise execute $\mathcal{G}$; *Step 3.* Terminate.
Simple Loop /* γ is a Boolean expression and $\mathcal{F}$ is an algorithm. */ **do** $\gamma \rightarrow \mathcal{F}$ **od**	*Step 1.* Evaluate the loop guard γ. If γ evaluates as **false,** then terminate. If γ evaluates as **true,** then go to Step 2. If γ is undefined, then crash; *Step 2.* Execute the loop body $\mathcal{F}$. When and if $\mathcal{F}$ terminates, go to Step 1.

his students. Since the Dijkstra-Gries programming notation is very simple, we enjoy the benefit of being free from the "language bias" that would be hard to avoid by using such programming languages as PL/I, C, or Pascal.

The subset of the Dijkstra-Gries instructions in Table 1 that we are using without modifications consists of the skip, the composition, the assignment, the simple IF, and the simple loop. Note, however, that instead of using the simple loop directly, we use its modification that we call the simple verifiable loop. Since all the loops we use are "hidden" inside generic algorithms, it is not necessary to understand simple verifiable loops in order to be proficient in using generic algorithms. Simple verifiable loops are described in Appendix A.

Our use of the term "pseudocode instruction" given in Table 2 is not quite common. We write pseudocode instructions in the form of specifications with the identifiers taken from the data structure of the overall algorithm (with new specification variables possibly added).[22] The only allowable difference in the treatment of the identifiers is that some variables from the overall data structure could be redeclared as constants within the scope of the instruction. It means that the instruction may not change the values of these variables during its execution. As a shorthand we assume that all the variables not explicitly occurring in the invariant and the postcondition of the instruction are regarded as constants, and they need not be explicitly redeclared.

On usage of terms "program" and "algorithm." Within the program development community the term "program" usually means "something that is written in a programming language and that can be converted by a compiler into executable code." The term "algorithm" is understood in its

Table 2 Pseudocode Instruction

Pseudocode Instruction	Behavior During Execution
/* α, β, φ, ψ are logical assertions and CON is an optional clause of the form **con** V, where V is a possibly empty list of variables from the overall data structure. The status of all the identifiers from V together with all variables not occurring in $\beta \wedge \psi$ is temporarily changed from "variable" to "constant" with the scope of the change limited to this instruction. After the instruction is completed, their former status is restored. */ |[CON; **constr** α; **inv** β; **pre** φ; **post** ψ]|	/* If any of **constr** α; **inv** β; **pre** φ; or **post** ψ is omitted, the respective clause is defined to be **true**. For inheriting the constraint of the overall specification we write **constr***. */ *Step 1.* Evaluate $\alpha \wedge \beta \wedge \varphi$ in the initial state. If it evaluates as **false** or is undefined, then crash. If it evaluates as **true,** then go to Step 2; *Step 2.* If there is a state such that: (a) the values of all the constants, of the identifiers from V and of all the variables not occurring in $\beta \wedge \psi$, are the same as in the initial state; (b) the resulting state satisfies $\alpha \wedge \beta \wedge \psi$, then choose any state satisfying (a) and (b) as the final state and terminate. Otherwise crash.

usual mathematical sense, i.e., "a sequence of instructions denoting some meaningful actions." An intermediate product of program development (which could be thought of as a mixture of program instructions and pseudocode) is usually referred to as "design."

In contrast, in the literature the term "program" is sometimes used loosely and may include what program developers could regard as "design." Recognizing this, we suggest calling all of the constructions composed from standard instructions and pseudocode instructions "algorithms," while reserving the term "programs" for algorithms that do not include the pseudocode instructions. It is then clear that "derivation of algorithms from specifications" refers to the development of both programs and design in the terminology of program developers.

Program derivation and the generic algorithms approach

The Dijkstra-Gries school of program derivation. The essence of the program development approach taught by the Dijkstra-Gries school consists of extracting a proved algorithm from a given specification in such a way that both the algorithm and the proof are developed in small increments with elements of the proof preceding the corresponding elements of the algorithm. Despite some more recent publications, classical Gries[2] is still the most definitive book on the subject, containing not only numerous techniques, examples, and anecdotes but also such advanced topics as dealing with partiality (see more on it in Appendix A). A later refinement of the methodology is known as "calculational."[3,4,23] By adding more techniques for manipulation with formulas, it shows how programs could be calculated from the specifications in a rigorous and elegant way.

Since the DO-loop is the most difficult programming element to prove, this school concentrates mostly on extracting "candidates" for the loop invariants from the specifications. Once a good "candidate" is found, the loop guard and the loop body are "calculated" around it. If the calculation is unsuccessful, another "candidate" will be sought. This methodology is very powerful. However, it is not free of disadvantages:

1. To efficiently utilize the methodology, one has to have a high level of mathematical skill. An ability to prove simple logical assertions may not be enough.
2. When reasoning about the program on the level of invariants, a great deal of programming intuition is lost. For instance, while looking into Kaldewaij's elegant solution of the maxsegsum problem, one could have a feeling that we are dealing with a recursively defined function. But it is not explicitly seen behind the manipulations with formulas.
3. As presently taught, the methodology requires application of the same detailed technique

again and again, though for different programs. In deriving programs, one can often see that the exact same thing was already done before, but there is no way given to "reuse" it. It is especially obvious with the "tail recursion" and "search by elimination" techniques discussed in Kaldewaij.[3]

Recently an interesting work by Bohorquez and Cardoso[24] addressed the second disadvantage by providing intuitive motivation for some of the techniques for extracting the loop invariants and by making them more general. However, this work did not address the first and third disadvantages.

Generic algorithms. We now describe the generic algorithms of our approach.

The approach. Our new methodology "deriving programs using generic algorithms" extends the Dijkstra-Gries methodology and is designed to overcome the three disadvantages listed above. It is based on the notion of a generic algorithm. A generic algorithm has the following features:

- A formal specification
- A formal proof that it satisfies this specification
- Generics

Our methodology consists of:

- Creating a collection of generic algorithms that cover most of the DO-loops occurring in programs
- Applying this collection to the program derivation process as described later

This methodology overcomes the previously mentioned disadvantages as follows:

1. Since generic algorithms are supplied with loop invariants and bound functions, the practitioner is freed from either looking for the loop invariants or proving that the loops terminate. Therefore, with the generic algorithms an ability to prove simple logical assertions may be sufficient.
2. Generic algorithms enhance the intuition. For instance, the maxsegsum problem is easily solved by an intuitively clear generic algorithm for computing recursive functions.
3. Generic algorithms allow reuse of both the design and the proofs. For instance, "tail recursion" and "search by elimination" techniques were converted into generic algorithms.

Although the features of a generic algorithm are not new, their combination, together with the way we apply the generic algorithms to the program derivation process, is new.

However, we would like to acknowledge a previous work (see Kieburtz and Shultis[25]) developing an approach in some aspects similar to ours, though quite different overall. We discuss the similarities and differences between the approaches in Appendix A.

Examples of generic algorithms. The first example of a generic algorithm is Star_Recursion shown in Figure 5.

The next generic algorithm, shown in Figure 6, illustrates the use of specification variables.

A library of generic algorithms. We have created a library of generic algorithms that implicitly contains many techniques of program derivation converted into generic algorithms, as well as generic algorithms not corresponding to a single such technique. Some of our generic algorithms were developed using the Dijkstra-Gries methodology, and the rest were derived using our extension of the Dijkstra-Gries methodology. So far we have about 30 algorithms in our library. On the basis of our development experience and by working through numerous examples in such classical textbooks as Gries,[2] Cohen,[4] and Kaldewaij,[3] we became convinced that our library covers most of the DO-loops a programmer could conceivably encounter in everyday work.

In Yakhnis, Farrell, and Shultz[26] we list the library of generic algorithms in Dijkstra-Gries notation. The work of converting them into other languages is underway. We present below a partial list of these generic algorithms:

1.0 Computation of recursive functions using DO-loops
  1.1 Simple recursive function
  1.2 Finite memory recursive function
  1.3 Star recursion

2.0 Searches
  2.1 Search by elimination
  2.2 Searches with quantifiers

Figure 5 Star recursion algorithm

```
|[Star_Recursion
/* Compute G(yzero) and put the result into r. G does not occur in the standard instructions. A function G takes a 𝕐 value
and creates an 𝕏 value. It satisfies the "star recursion" property: G(y) = G(tail(y)) ☆ head(y), where ☆ is a binary
associative operation and tail and head are functions declared below. */

gen          𝕐, 𝕏                                   /* sets */

spec         G : 𝕐 → 𝕏
             BOUND : 𝕐 → ℕ                          /* ℕ is the set of all nonnegative integers */

con          base : {y: 𝕐|low_bound(y)}→𝕏            /* computes the function G for y ∈ 𝕐 with low_bound(y) = true */
             low_bound : 𝕐 → 𝔹                       /* 𝔹 = {true, false} */
             tail : {y: 𝕐|¬low_bound(y)}→𝕐
             ☆ : 𝕏 × 𝕏 → 𝕏
             idstar : 𝕏                             /* identity value for ☆ operation */
             head : {y: 𝕐|¬low_bound(y)}→𝕏
             yzero : 𝕐                              /* input */

var          r : 𝕏                                  /* output */

constraint
☆ is a binary associative operation with an identity value
∧
idstar is the identity value of ☆
∧
(∀y:𝕐 •                                             /* star recursion properties */
  low_bound(y) ⇒ G(y) = base(y)
  ∧
  ¬low_bound(y) ⇒ G(y) = G(tail(y)) ☆ head(y)
  ∧
  low_bound(y) ⇔ BOUND(y) = 0
  ∧
  ¬low_bound(y) ⇒ BOUND(tail(y)) ≺ BOUND(y))

postcondition   r = G(yzero)

algorithm
|[

var      y : 𝕐                                      /* work variable */

  y := yzero;                                       /* initialize work variable */
  r := idstar;                                      /* initialize output value */
  do ¬low_bound(y) →

     invariant G(yzero) = G(y) ☆ r

     bound function BOUND(y)

     r := head(y) ☆ r;                              /* accumulate new r */
     y := tail(y)                                   /* decrement loop counter */
  od;
  r := base(y) ☆ r                                  /* complete accumulation of output value */
]|
]|
```

Figure 6 Generic algorithm with specification variables

```
|[Action_Unbounded_Linear_Search_Strict
/* For every element x in a given nonempty ordered finite set SETAB, with the exception of the greatest element, we
would like to "do something," i.e., perform unspecified actions making a property P(x) of the program variables true.
Simultaneously, we would like to find the greatest element in SETAB. We will know that x is the greatest element when a
given Boolean function high_bound(x) returns "true." */

gen            SETAB                                   /* set */

spec con       < : SETAB↔SETAB                         /* ↔ is the binary relation symbol. */
               B : SETAB

spec var       P : {x: SETAB | x ≠ B} → 𝔹               /* P(x) is defined for all x such that x is in SETAB and x ≠ B */

   con         a : SETAB
               next : {x: SETAB | x ≠ B} → SETAB       /* next(x) is defined for all x ≠ B */
   high_bound : SETAB → 𝔹

var            i : SETAB

constraint

SETAB is totally ordered by <
∧
a is the smallest element of SETAB in respect to <
∧
B is the greatest element of SETAB in respect to <
∧
B is the unique element of SETAB satisfying high_bound(x) = true
∧
next is the successor function on SETAB in respect to <
/* If x is in SETAB and x ≠ B then next(x) is the smallest element of SETAB that is greater than x in respect to <. */

postcondition
/* If α and β are statements describing some properties of x, then "(∀x : 𝕏 | α · β)" means "for all such x from the set 𝕏
that α is satisfied, β is satisfied as well." */

i = B ∧ (∀x : SETAB | x < B · P(x))

algorithm
i := a;
do     ¬high_bound(i) →
  invariant i ∈ SETAB ∧ (∀x : SETAB | x < i · P(x))
  bound function #{x : SETAB | i < x ≤ B}     /* The number of elements in SETAB that are greater than i and less or
                                                 equal than B. */
  |[inv (∀x : SETAB | x < i · P(x)); post P(i))]|;
  i := next(i);
od
]|
```

Deriving programs using generic algorithms. The following subsections show the use of our approach in program development.

The program derivation process. With the advent of the library of generic algorithms the program derivation process could be ideally represented as follows. Given a formal specification we must first check whether it is sensible. If it is, view it as an algorithm consisting of a unique pseudocode instruction. The process of derivation consists of several stages, each resulting in an algorithm having more detail than the previous ones. We may stop either when the algorithm becomes a pro-

gram or before that. In either case the resulting algorithm will satisfy the initial specification. Each of those stages consists of one or more iterations of the following steps:

1. Choose a pseudocode instruction from the algorithm created in the previous stage.
2. Choose a generic algorithm in the library that might be doing similar work.
3. Adapt the generic algorithm by replacing some of its identifiers with the conceptually similar constructions based on the identifiers of the pseudocode instruction. Adapt the specification of the generic algorithm by the same replacement of identifiers. Below we explain this step in more detail.
4. Justify that the adapted specification from Step 3 correctly implements the specification of the pseudocode instruction. We explain later how to do it in a simple three-step procedure. If the justification fails, go back to Step 2.
5. Replace the pseudocode instruction in the algorithm created in the previous stage by the algorithm from Step 3.

In the following two subsections we illustrate Steps 3 and 4 by deriving an algorithm factorial from the specification Factorial_Spec. We can arrive at Step 3 with relative ease. It is easy to see that Factorial_Spec is sensible (i.e., there is at least one initial state). Step 1 is trivial since we have only one pseudocode instruction, namely Factorial_Spec itself. At Step 2 we look at the formula $n! = (n - 1)! * n$ and see a resemblance to the star recursion formula $G(y) = G(\mathit{tail}(y)) \star \mathit{head}(y)$ from Star_Recursion. Thus we choose the latter for Step 3.

Replacing identifiers in the generic algorithm: Clarifying Step 3. As already mentioned, we adapt the generic algorithm by replacing some of its object identifiers by their *analogs*, i.e., conceptually similar constructions based on the identifiers of the pseudocode instruction. The adapted algorithm is intended to work with the states defined by the data structure of the pseudocode instruction (with, possibly, some work variables added as described below). As we shall see, some of the adapted algorithms should be rejected even before going to Step 4. We start from the following two questions:

1. Which object identifiers must have analogs?
2. What is to be done with the identifiers that do not have analogs?

First, all the specification identifiers, constants, and input variables must have analogs, otherwise the adapted algorithm should be rejected. In our example, a comparison between the factorial formula and the star recursion formula yields "!" as the analog of G and "$*$" as the analog of "$\star$". A comparison between the postconditions yields r as the analog of r and *nzero* as the analog of *yzero*. We defer finding the analogs of the rest of the identifiers until we answer the second question.

Second, some of the noninput variables may not have analogs. For example, the work variables never have analogs. Recall that the "work" variables are those variables that do not appear in the specification of the algorithm but only in the body of the algorithm. In our example, y is the only work variable in Star_Recursion, and it does not have any obvious analog in terms of Factorial_Spec. Answering the second question, we say that each noninput variable x not having an analog is retained in the adapted algorithm as a work variable. However, if the specification of the pseudocode instruction already has an identifier with the same name, x should be renamed in order to avoid the clash. Later we will discuss which types may be assigned to the variables that we retain.

We would like to introduce the notion of a *data structure translating map* which is a convenient tool for analyzing the adaptation of generic algorithms. Such a tool maps every object identifier of the generic algorithm having an analog into this analog, and it maps every other object identifier into itself. In addition, it maps the generic type identifiers into some type constructions based on the type identifiers of the pseudocode instruction. Since a formal discussion of how to map the generic type identifiers is beyond the scope of this paper (it is based on the notion of "signature morphism" [see Appendix A] and is not needed for most practical applications), we illustrate the mapping of types on our example.

Comparing the declarations $!:\mathbb{N}\to\mathbb{N}$ and $G:\mathbb{Y}\to\mathbb{X}$, we guess that it is natural to map both $\mathbb{Y}$ and $\mathbb{X}$ into $\mathbb{N}$. Designating the data structure translating map for our example as h, we see that $h(\mathbb{Y}) = \mathbb{N}$, $h(\mathbb{X}) = \mathbb{N}$, $h(G) = !$, $h(\star) = *$, $h(r) = r$, $h(\mathit{yzero}) = \mathit{nzero}$ and $h(y) = y$. The data structure translating map could be easily extended to mapping formu-

las: if φ is a formula, then h(φ) is the result of replacing every identifier x in φ by h(x). Using this we can extend h to all type constructions, e.g., the type of *tail* is $\{y:\mathbb{Y}|\neg$*low_bound*$(y)\} \rightarrow \mathbb{Y}$. Later we will learn that the analog of $\neg$*low_bound*(y) is $y \neq 0$. Thus, $h(\{y:\mathbb{Y}|\neg$*low_bound*$(y)\}\rightarrow\mathbb{Y})$ is $\{y:N|y\neq 0\}\rightarrow\mathbb{N}$, which is the set of all functions taking positive integers and returning nonnegative integers.

As was already mentioned, the adapted algorithm operates on the states defined by the data structure of the pseudocode instruction augmented by some work variables. Thus, for every identifier of the generic algorithm x having analog h(x), the type of h(x) in the adapted algorithm is determined by the pseudocode instruction. Given an object identifier $x:\mathbb{V}$ in the generic algorithm, the type of h(x) in the adapted algorithm may be distinct from $h(\mathbb{V})$. For example, *yzero* is declared as $yzero:\mathbb{Y}$, h(*yzero*) = *nzero*, $h(\mathbb{Y}) = \mathbb{N}$ but *nzero* is declared as *nzero*: $\mathbb{Z}$ in Factorial_Spec.

In the adapted algorithm, we may assign any types to the work variables as long as the formula Keeping the Variables within Types defined below is true. This formula is designed in such a way that if a work variable is assigned a type $\mathbb{W}$ according to the formula, it may be assigned any type containing $\mathbb{W}$. The other restriction on types may be imposed by the programming language. We assume now that we assigned some types to the work variables. Adding the corresponding declarations to the data structure of the pseudocode instruction, we form the *extended data structure*. In our example, we assign to y the type $\mathbb{Z}$.

Now, suppose that h is a data structure translating map, and suppose that *Constr*, *Inv*, and *Pre* are, respectively, the constraint, the invariant, and the precondition of the pseudocode instruction and *CONSTR* is the constraint of the generic algorithm. We designate the collection of all object identifiers of the generic algorithm as *ALL*, the collection of all specification identifiers, constants, and input variables of the generic algorithm as *INPUT*, and the collection of all variables of the generic algorithm as *VAR*.

Also, for a collection S of object identifiers of the generic algorithm, let *Gen_Type_of*(S) be the conjunction of all the statements of the form $x \in \mathbb{X}$, where x is in S and $\mathbb{X}$ is its type in the generic algorithm. Let *Pseudo_Type_of*(S) be the conjunction of all the statements of the form $h(x) \in \mathbb{Y}$, where x is in S and $\mathbb{Y}$ is the type of h(x) in terms of extended data structure of the pseudocode instruction. We say that h "respects the types" if the following two formulas are true:

- Initiating the Identifiers:
  (*Constr* $\wedge$ *Inv* $\wedge$ *Pre* $\wedge$ *Pseudo_Type_of*(*ALL*))
  $\Rightarrow$ h(*Gen_Type_of*(*INPUT*));
- Keeping the Variables within Types:
  (h(*CONSTR*) $\wedge$ h(*Gen_Type_of*(*ALL*)))
  $\Rightarrow$ *Pseudo_Type_of*(*VAR*).

Respecting the types is a necessary condition for correctness of our proof of the validity of the method (see Appendix A). Therefore, we reject an adapted generic algorithm if the corresponding data structure translating map does not respect the types.

Returning to our example, let us verify that the part of h that we have defined so far does not violate the "respecting the types" property. As an example, to check the property for *yzero*, it is sufficient to show that (*Constr* $\wedge$ *Inv* $\wedge$ *Pre* $\wedge$ *nzero* $\in \mathbb{Z}$) $\Rightarrow$ *nzero* $\in \mathbb{N}$. (Since the identifiers may be interdependent, proving "respecting the types" in isolation may not always work.) This is trivial since *Constr* includes a conjunct *nzero* ≥ 0. In order to check the property for r, it is sufficient to show that (h(*CONSTR*) $\wedge$ $r \in \mathbb{N}$) $\Rightarrow$ $r \in \mathbb{Z}$. This is trivial since $\mathbb{N}$ is a subset of $\mathbb{Z}$. Let us show that the declaration $y:\mathbb{Z}$ does not violate the property of respecting the types. It is sufficient to show that (h(*CONSTR*) $\wedge$ $y \in \mathbb{N}$) $\Rightarrow$ $y \in \mathbb{Z}$. Again, this is trivial. According to our experience, similar proofs are almost always trivial.

The following shortcut often could be used for variables. If $x:\mathbb{V}$ is the declaration of a variable in the generic algorithm and if $h(x):\mathbb{W}$ is the declaration of h(x) in the extended data structure of the pseudocode instruction, then Keeping the Variables within Types is satisfied for x if $h(\mathbb{V})$ is a subset of $\mathbb{W}$.

Let us consider now what will replace the identifier *tail*. Comparing the equations for G and !, the best candidate for the replacement is the function that takes an integer value n and returns n − 1. However, the data structure for Factorial_Spec does not have a name associated with this function. Though mathematically valid, adding a new identifier associated with this function to Factorial_Spec would

violate a programming principle, formulated in Gries,[2] stating that in the process of derivation one should not add new identifiers, unless absolutely necessary. In our approach the only legitimate ways to add new identifiers is via "retaining variables" from the generic algorithm.

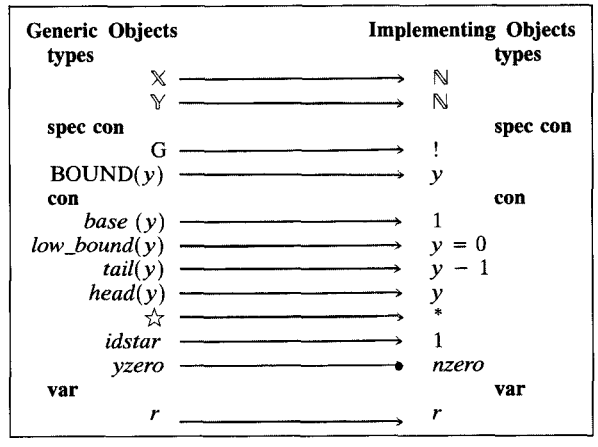
One way to deal with the situation is to use function constructions (see Appendix A). However, since their usage assumes the knowledge of λ-notation, we give here an alternative method involving the use of abbreviations. An abbreviation is a temporary name assigned to a function with the agreement that instead of using this name directly in the algorithm, only the value it returns when applied to arguments will be used. As an example, let us assign the name "*minus_one*" to the function that takes an integer value n and returns n − 1. We can define now h(*tail*) = *minus_one*. However, when we replace an expression *tail*(*y*), instead of replacing it with *minus_one*(*y*), we replace it with *y* − 1. If there is another expression involving *tail*, say, *tail*(*z*), we replace it with *z* − 1.

Similarly, we replace *head*(*y*) by *y*. Now we have exhausted the information we can extract from comparing n! = (n − 1)! * n and G(y) = G(*tail*(y)) ☆ *head*(y). Consider ¬*low_bound*(y) ⇒ G(y) = G(*tail*(y)) ☆ *head*(y). By replacing everything we can replace so far, we get ¬*low_bound*(y) ⇒ y! = (y − 1)! * y. Since the formula y! = (y − 1)! holds for all y > 0, ¬*low_bound*(y) corresponds to y > 0, and thus *low_bound* corresponds to y = 0. Let us check the "respecting the types" property for *tail*. We must show (*Constr* ∧ *Inv* ∧ *Pre* ∧ *minus_one* ∈ {y: ℤ|y > 0} → ℤ) ⇒ *minus_one* ∈ {y:ℕ|y > 0} → ℕ. The consequence of this implication directly follows from the definition of *minus_one*. We leave the rest to the reader.

Replacing the identifiers and expressions from Star_Recursion by those from Factorial_Spec is illustrated in Figure 7. The more mathematically precise data structure translating map is illustrated later in Figure 8 in Appendix A. The arrows point from the objects being replaced to the replacing objects.

Let *SC*, *SV*, *C*, and *V* be, respectively, the collections of specification constants, specification variables, constants, and variables from the pseudocode instruction, and let *SC'*, *SV'*, *C'*, and *V'* be their respective counterparts from the specification of the generic algorithm from Step 2.

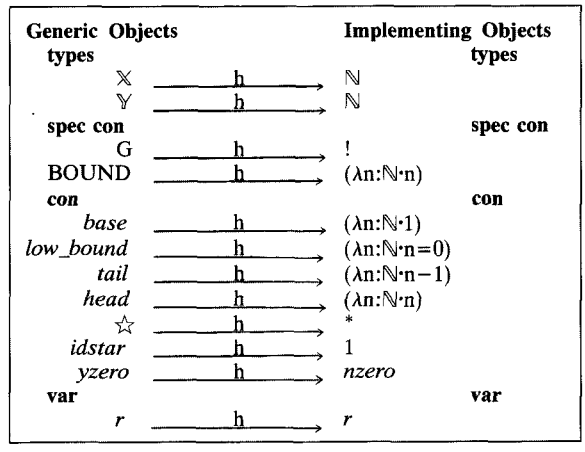
Figure 7 Replacing the identifiers and expressions from Star_Recursion by those from Factorial_Spec



We summarize now our requirements on a data structure translating map h:

1. Each specification constant from *SC'* must be mapped to a construction that may have only occurrences of identifiers from *SC* or *C*.
2. Each constant from *C'* must be mapped to a construction that may have only occurrences of identifiers from *C*.
3. Each specification variable from *SV'* must be mapped to a construction with no occurrences of variables from h(*V'*).
4. Each input variable from *V'* must be mapped to an input variable from *V*. Noninput varia-

Figure 8 The data structure translating map h from Star_Recursion to Factorial_Spec



bles are mapped into noninput variables. Different variables must be mapped into different variables.
5. A function may be replaced only by a function with the same number of arguments.
6. h respects the types.

If any one of the conditions is not satisfied, we reject h and the corresponding adapted algorithm. The conditions ensure that the correctness proof for the generic algorithm is still valid after replacing the identifiers via a data structure translating map (see Appendix A). However, if partial functions are used, the data structure translating map should satisfy an additional condition, formulated in Appendix A. Those concerned with the use of partial functions should also read Appendix A.

The data structure translating maps are related to answering the following question: When is it possible to implement a collection of types and operations by another collection of types and operations? Such an implementation is sometimes called "data reification." It is discussed in Jones,[12] Clement,[27] and other works. However, there is a subtle difference between the data structure translating maps and data reification. Whereas the latter starts from a more abstract structure and seeks to implement it by a more concrete structure in order to take advantage of the more concrete types and operations, the former starts from a more concrete structure and seeks to take advantage of the algorithm formulated in terms of a more abstract structure. Since the data structure translating maps have to accommodate the separation of the identifiers into four distinct classes and since we treat the partial function differently, the notion of data reification from Jones[12] is not sufficient for our purposes.

Implementing one specification with another specification: Clarifying Step 4. With the above clarification for Step 3, we may explain the procedure in Step 4. Suppose that *Constr*, *Inv*, *Pre*, and *Post* are, respectively, the constraint, the invariant, the precondition, and the postcondition from the pseudocode instruction. Assume further that *Constr'*, *Inv'*, *Pre'*, and *Post'* are their respective counterparts from the adapted specification of Step 3.

In order to be assured that the adapted specification from Step 3 correctly implements the specification of the pseudocode instruction, it is sufficient to verify the correctness of the following three formulas. Here we refer to the pseudocode instruction as "old" and to the adapted specification as "new":

1. Establishing the new initial extended constraint, invariant, and precondition:

$$(Constr \wedge Inv \wedge Pre \wedge Pseudo\_Type\_of(ALL)) \Rightarrow (Constr' \wedge Inv' \wedge Pre')$$

2. Maintaining the old extended constraint:

$$(Constr' \wedge \mathrm{h}(Gen\_Type\_of(ALL)) \Rightarrow Constr$$

3. Establishing the old postcondition and reestablishing the old invariant:

$$(Constr' \wedge Inv' \wedge Post' \wedge \mathrm{h}(Gen\_Type\_of(ALL))) \Rightarrow (Inv \wedge Post)$$

The data structure translating maps (together with the above formulas) can be used for mapping from any given specification, not just from the specification of a generic algorithm. Thus, the material in this section is related to answering the question: When does one formal specification implement another? Such an implementation is sometimes called "refinement." It is discussed in Gries,[2] Morgan,[15] Woodcock[28] and many other works. Our case differs from the traditional investigations of these questions in the separation of the object identifiers into four classes and (for the sequential case) usage of the constraints.

Derivation of Factorial. In order to finish the derivation of Factorial, we have to prove correctness of the three formulas from the previous discussion. We only do it for the third formula, leaving the rest to the reader.

Let us construct the adapted constraint using the data structure translating map from Figure 7:

$$(* \text{ is a binary associative operation with an identity}) \wedge (1 \text{ is the identity value of } *)$$

$$\wedge$$

$$(\forall y{:}\mathbb{N} \cdot (y = 0 \Rightarrow y! = 1) \wedge (y \neq 0 \Rightarrow y! = (y - 1)! * y) \wedge (y = 0 \Leftrightarrow y = 0) \wedge (y \neq 0 \Rightarrow y - 1 < y))$$

By throwing out all trivially true conjuncts, we get

$(\forall y{:}\mathbb{N}\cdot(y=0\Rightarrow y!=1)\wedge(y\neq 0\Rightarrow y!=(y-1)!*y))$

Finally, we are required to show that

$$((\forall y{:}\mathbb{N}\cdot(y=0\Rightarrow y!=1)\wedge(y\neq 0\Rightarrow y! = (y-1)!*y))\wedge r\ nzero!)\Rightarrow (\forall n{:}\mathbb{N}\cdot(n=0\Rightarrow n!=1)\wedge(n\neq 0\Rightarrow n!=(n-1)!*n))\wedge r = nzero!$$

This is obvious. The validity of the other two formulas is just as easy to establish. Now we can be assured that the adapted algorithm satisfies Factorial_Spec. We list the adapted algorithm below.

```
|[Factorial
Factorial_Spec
|[
var     y:ℤ        /* work variable */

y := nzero;        /* initialize work variable */
r := 1;            /* initialize output value */
do y ≠ 0 →
   r:= y * r;      /* accumulate new r */
   y := y - 1;     /* decrement loop counter */
od;
/* r = 1 * r; when we inherited this assignment
from the respective generic algorithm it became
trivial. Thus we may exclude it from the code. */

]|
]|/* end of Factorial */
```

Concluding remarks

The generic algorithm approach presented here contains several innovations in the application of formal methods to software development. The main advance is the notion and use of generic algorithms, including their properties and the rules for how they can be applied.

Generic algorithms are used in the program derivation process rather than the various techniques for finding the loop invariants. With the generic algorithm method for deriving programs, programmers have at their disposal a library of reusable building blocks that they can use to build a program whose correctness is proved. They must understand the concepts of formal specification, but not the difficult techniques of deriving loops. The use of generic algorithms enhances programming intuition and allows for the reuse of both design and proofs. Experience in our pilot efforts in VM development and teaching classes in this approach support our contention that this method can be used by programmers who are not experts at program proofs or formal derivation.

We introduce constraints and specification variables within the framework of program derivation, and we extend Dijkstra's "weakest precondition" predicate transformer, enabling it to work with constraints, pseudocode instructions, and simple verifiable loops (see Appendix B).

Acknowledgments

Vlad Yakhnis would like to express his gratitude to Professor David Gries for teaching him the program derivation methodology and for his encouragement and support. All the authors are deeply grateful to Alex Yakhnis for suggesting numerous improvements in the final draft of this paper. We are indebted to Tom Vail for reading the first draft of this paper and making many valuable suggestions. Many thanks to Michael Walker for several helpful suggestions. Finally, we are indebted to the anonymous referees for their constructive criticism of the first and second draft of this paper.

Appendix A: Technical details

Function constructions. Here we define function constructions using the standard typed λ-notation. Let us give a brief definition of the λ-notation. Suppose that E is an expression of type $\mathbb{X}$. The notation $\lambda y{:}\mathbb{Y}\cdot E$ designates the following function taking a $\mathbb{Y}$-value and returning an $\mathbb{X}$-value: Given a value w from $\mathbb{Y}$, $(\lambda y{:}\mathbb{Y}\cdot E)(w)$ is computed by: (a) replacing all occurrences of y in E by w; and (b) returning the value of the resulting expression.

Our notation for replacing y in E by w is $E(y\leftarrow w)$. Thus $(\lambda y{:}\mathbb{Y}\cdot E)(w)$ is defined as $E(y\leftarrow w)$, e.g., $(\lambda y{:}\mathbb{N}\cdot(y-1))(2) = 1$ and $(\lambda y{:}\mathbb{N}\cdot(y=0))(1)$ = **false**.

Let us take advantage of function constructions in describing the data structure translating map from Star_Recursion to Factorial_Spec. This map is given in Figure 8.

Conventions for partial functions and uninitialized variables. Using expressions that may be undefined on some states is the way of life for prac-

ticing programmers (and often that is precisely what makes this life miserable). Undefined expressions stem from two distinct reasons: usage of partial functions and usage of uninitialized variables. Unfortunately, first order logic, the easiest tool for program correctness proofs, does not treat either of these subjects (e.g., see Shoenfield[29]).

Though some works treat partiality at length (e.g., see Breu[30]), very few discuss it in the context of program correctness proofs. We distinguish three influential approaches, Tucker and Zucker,[31] VDM (e.g., Jones[12]), and Gries.[2] Tucker and Zucker[31] may offer the first truly exhaustive treatment of abstract data types with error state semantics (i.e., allowing uninitialized variables), but it is not addressed to practicing programmers. VDM, while intended for practicing programmers, introduces a three-valued logic and thus goes beyond first order logic. Finally, the Gries approach[2] provides a treatment of partiality sufficient for our purpose while remaining in the framework of first order logic. Our approach below is based on Gries with some modifications.

We eliminate the problem of undefined variables by assuming that algorithms assign values to all variables at the beginning of execution. This is true for our generic algorithms. For dealing with undefined expressions we introduce an operator Def, converting each expression into a Boolean expression evaluating as **true** if the original expression is undefined and **false** otherwise. For an expression consisting of a single identifier x, Def(x) is equal to

- **true**, if x is a constant
- $x \in \mathbb{X}$, if x is a variable declared as x:$\mathbb{X}$

Before defining Def for all expressions, we introduce explicit domains for functions. For a partial function f:$\mathbb{X} \mapsto \mathbb{Y}$ the *domain* of f is a total Boolean-valued function $\mathrm{Dom}(f)$: $\mathbb{X} \to \mathbb{B}$ such that for every x in $\mathbb{X}$ $\mathrm{Dom}(f)(\mathrm{x})$ = **true** if $f(\mathrm{x})$ is defined and $\mathrm{Dom}(f)(\mathrm{x})$ = **false** otherwise. We consider only such functions for which there is a formula $\psi(\mathrm{x})$ written in the language of the data structure but without occurrences of partial functions and such that $(\forall \mathrm{x}{:}\mathbb{X} \cdot \mathrm{Dom}(f)(\mathrm{x}) \Leftrightarrow \psi(\mathrm{x}))$. We place $(\forall \mathrm{x}{:}\mathbb{X} \cdot \mathrm{Dom}(f)(\mathrm{x}) \Leftrightarrow \psi(\mathrm{x}))$ in the constraint. As a shortcut we allow declaring f as f: $\{\mathrm{x}{:}\mathbb{X} | \psi(\mathrm{x})\} \to \mathbb{Y}$ instead of f:$\mathbb{X} \mapsto \mathbb{Y}$, while omitting the explicit definition of $\mathrm{Dom}(f)$ from the constraint. If g is a total function declared as g:$\mathbb{X} \to \mathbb{Y}$, then $\mathrm{Dom}(g)(\mathrm{x})$ = **true** for all $\mathrm{x} \in \mathbb{X}$.

For example, in the specification for Star_Recursion in Appendix B, the function *base* is declared as *base* : $\{\mathrm{y}{:}\mathbb{Y} | \mathit{low\_bound}(\mathrm{y})\} \to \mathbb{X}$. This is an abbreviation for declaring *base* as *base*: $\mathbb{Y} \mapsto \mathbb{X}$ and adding $(\forall \mathrm{y}{:}\mathbb{Y} \cdot \mathrm{Dom}(\mathit{base})(\mathrm{y}) \Leftrightarrow \mathit{low\_bound}(\mathrm{y}))$ to the constraint.

A generalization of the domains for functions and partial functions of several variables is straightforward. Finally, we illustrate how the operator Def is defined on arbitrary expressions by means of the following example. Let f be a binary function or partial function and A and B be some expressions. Then $\mathrm{Def}(f(A,B))$ is equal to $\mathrm{Dom}(f)(A,B) \wedge \mathrm{Def}(A) \wedge \mathrm{Def}(B)$. In addition, Def is used to explicitly define domains of the λ-expressions, e.g., $\mathrm{Dom}(\lambda \mathrm{y}{:}\mathbb{Y} \cdot E) = (\lambda \mathrm{y}{:}\mathbb{Y} \cdot \mathrm{Def}(E))$.

Having explicitly defined the operator Def, it is possible for a given algorithm to write a statement saying "the algorithm never attempts to evaluate an undefined expression." A formal algorithm correctness assertion that implies the above statement is discussed in Appendix B.

Using simple verifiable loops in generic algorithms. The simple verifiable loop is the Dijkstra-Gries simple loop augmented by the loop invariant and the bound function and is based on the famous "checklist for understanding a loop" from Gries.[2] Although the simple verifiable loop is widely used in the framework of program derivation, it is not recognized as an instruction in its own right. We extend the ideas of Gries by recognizing that the simple loop is needed mostly as a stepping stone for understanding the simple verifiable loop and that the latter is an instruction deserving a separate proof rule. We give such a rule in Appendix B. In our generic algorithms we use simple verifiable loops instead of simple loops (see Table 3).

Comparing generic algorithms with schemes from Kieburtz and Shultis. In the pioneering work of Kieburtz and Shultis,[25] a version of generic algorithms (called "schemes") was used within the framework of functional programming. The purpose was to convert a given recursive definition of a function into an efficient DO-loop and to prove that the function and the DO-loop compute identical values.

Table 3 Using simple verifiable loops

Simple Verifiable Loop	Behavior During Execution
/* γ is a Boolean expression, φ is a logical assertion, E is an integer-valued specification expression and $\mathcal{F}$ is an algorithm. It is established that φ is an invariant of $\mathcal{F}$ and that E is a bound function. */	/* The following must be proved beforehand: $\{\varphi\}\ \mathcal{F}\{\varphi\} \wedge (\varphi \Rightarrow \mathrm{Def}(\gamma))$, i.e., φ is an invariant of $\mathcal{F}$, and $(\varphi \Rightarrow E \geq 0) \wedge \{E = X\}\ \mathcal{F}\ \{E < X\}$, i.e., if φ is true, then E is nonnegative and every execution of $\mathcal{F}$ decreases E at least by 1, i.e., E is a bound function. */
do $\gamma \rightarrow$ **invariant** φ **bound function** E $\mathcal{F}$ **od**	*Step 1.* Evaluate the invariant φ. If φ evaluates as **false**, then crash. If φ evaluates as **true,** then go to Step 2; *Step 2.* Evaluate the loop guard γ. If γ evaluates as **false,** then terminate. If γ evaluates as **true**, then go to Step 3, If γ is undefined, then crash; *Step 3.* Execute the loop body $\mathcal{F}$. When and if $\mathcal{F}$ terminates, go to Step 2.

In a sense, the aforementioned recursive definition of a function could be viewed as a form of specification for the loop. Since they were proved to be computationally equivalent and since both contained generics, a degree of similarity with generic algorithms exists. Moreover, one of the schemes used in Kieburtz and Shultis is very similar to our Star_Recursion, though not identical. This scheme is both more and less general than Star_Recursion. On one hand, its constraint on $\star$ is more general than associativity; on the other hand, the scheme requires *base* to be a constant function equal to *idstar*.

Nonetheless, there are significant differences between the approaches:

- Our purpose is finding algorithms satisfying specifications, whereas the purpose of Kieburtz and Shultis is finding an efficient way to execute programs written in functional style.
- In many cases a specification does not have an obvious rendition into a recursive definition. In fact, converting a specification into a recursive definition goes a long way toward providing a solution.
- In contrast to our generic algorithms, the schemes from Kieburtz and Shultis are not supplied with preconditions. This means that the question of termination was left open.
- Most of our generic algorithms are not covered by the schemes from Kieburtz and Shultis.

In spite of the differences, Kieburtz and Shultis may be useful for us since in the future we may convert some of their schemes into generic algorithms by supplying their loops with loop invariants and bound functions.

Partial functions and data structure translating maps. Earlier in this appendix we described the notion of an explicit domain $\mathrm{Dom}(f)$ for a partial function f. Assume that f is a unary function. Then $\mathrm{Dom}(f)(\mathrm{x})$ is a Boolean expression that may have occurrences of some identifiers from the data structure. If $f{:}\mathbb{X} \leftrightarrow \mathbb{Y}$ is an identifier in the specification of the generic algorithm and h is a data structure translating map from the generic algorithm to the specification of the pseudocode instruction, we may have two distinct Boolean expressions, $\mathrm{h}(\mathrm{Dom}(f)(\mathrm{x}))$ and $\mathrm{Dom}(\mathrm{h}(f))(\mathrm{x})$. However, the "respecting the types" property of h allows us to use $\mathrm{h}(\mathrm{Dom}(f)(\mathrm{x}))$ instead of $\mathrm{Dom}(\mathrm{h}(f))(\mathrm{x})$. Therefore, in terms of Wirsing[19] and Breu,[30] h can be regarded as a "signature morphism."

A sketch of proof of the validity of the approach. The more difficult part of proving the validity of the approach is showing that the adapted generic algorithm satisfies its adapted specification. With this out of the way, the formulas given earlier in the subsection on implementing one specification with another specification make the rest obvious. Here we give a sketch of this more difficult part of the proof.

Assume that $\mathcal{F}$ is the generic algorithm and h is the data structure translating map. Assume further that *CONSTR*, *INV*, *PRE*, and *POST* are, respec-

tively, the constraint, the invariant, the precondition, and the postcondition of $\mathcal{F}$. Since in the properties of generic algorithms we included a requirement that they are proved to be correct, we can reformulate the problem as the question: given a true formula $\{INV \wedge PRE\}\ \mathcal{F}/CONSTR\ \{INV \wedge POST\}$, how can we be sure that $\{h(INV) \wedge h(PRE)\}\ h(\mathcal{F})/h(CONSTR)\ \{h(INV) \wedge h(POST)\}$ is true as well?

Using wpc ("weakest precondition with constraints") (see Appendix B) we can transform $\{INV \wedge PRE\}\ \mathcal{F}/CONSTR\ \{INV \wedge POST\}$ into a first order formula (actually, several formulas may be involved as explained in Appendix B, but it would not change the reasoning):

$$INV \wedge PRE \Rightarrow \text{wpc}(\mathcal{F}/CONSTR, INV \wedge POST)$$

In this formula all the identifiers that are variables are regarded as first order variables (and are closed under universal quantification), and other identifiers are regarded as "nonlogical symbols."[29] Let us designate the formula as ψ. Replacing all the identifiers in ψ using the map h, we get a formula $h(\psi)$. Properties 4 and 5 of h ensure that $h(\psi)$ is true whenever ψ is true. The problem is that, in general, $h(\psi)$ may not be identical to

$$h(INV) \wedge h(PRE) \Rightarrow \text{wpc}(h(\mathcal{F})/h(CONSTR), h(INV) \wedge h(POST)).$$

Now, employing property 6 of h we can use $h(\text{Dom}(f)(x))$ in lieu of $\text{Dom}(h(f))(x)$. With this in mind and the use of properties 4, 5, and 6 of h, by induction on definition of wpc it is easy to show that $h(\psi)$ is indeed identical to

$$h(INV) \wedge h(PRE) \Rightarrow \text{wpc}(h(\mathcal{F})/h(CONSTR), h(INV) \wedge h(POST)).$$

This completes the proof.

Appendix B: Extending Dijkstra's weakest precondition to constraints and pseudocode

Weakest precondition without constraints. We first extend the Dijkstra weakest precondition wp to simple verifiable loops and pseudocode instructions. We use the Gries treatment of partiality[2] while relying on our definition of the operator Def. Here "$\triangleq$" means "equal by definition." We assume that the logical assertions (like Q or φ) are everywhere defined, whereas the expressions (like E or γ) may be nontotal.

1. Assignment: $\text{wp}(x := E, Q) \triangleq Q(x \leftarrow E) \wedge \text{Def}(E)$
2. Composition: $\text{wp}(\mathcal{F}; \mathcal{G}, Q) \triangleq \text{wp}(\mathcal{F}; \text{wp}(\mathcal{G}, Q))$
3. Simple IF: Let $\mathcal{P}$ be the following program:

   if $\gamma \rightarrow \mathcal{F}$
   $\square\ \neg\gamma \rightarrow \mathcal{G}$
   fi

   then

   $$\text{wp}(\mathcal{P}, Q) \triangleq \text{Def}(\gamma) \wedge (\gamma \Rightarrow \text{wp}(\mathcal{F}, Q)) \wedge (\neg\gamma \Rightarrow \text{wp}(\mathcal{G}, Q))$$

4. Simple verifiable loop: Let $\mathcal{P}$ be the following program:

   do $\gamma \rightarrow$
   invariant φ
   bound function E
   $\mathcal{F}$
   od

   Recall the properties of γ, φ, E, and $\mathcal{F}$.

   $\{\varphi\}\ \mathcal{F}\ \{\varphi\} \wedge (\varphi \Rightarrow \text{Def}(\gamma))$, i.e., φ is an invariant of $\mathcal{F}$, and $(\varphi \Rightarrow E \geq 0) \wedge \{E = X\}\ \mathcal{F}\ \{E < X\}$ (where X occurs neither in $\mathcal{F}$ nor in E), i.e., if φ is true, then E is nonnegative and every execution of $\mathcal{F}$ decreases E at least by 1, i.e., E is a bound function.

   Then

   $$\begin{array}{lll} \text{wp}(\mathcal{P}, Q) & \triangleq \varphi & \text{if } \varphi \wedge \neg\gamma \Rightarrow Q \\ & \triangleq \textbf{false} & \text{otherwise.} \end{array}$$

5. Pseudocode instruction: Let α, β, φ, ψ be logical assertions and CON be an optimal clause of the form **con** V, where V is a possibly empty list of variables from the overall data structure, and let $\mathcal{P}$ be the following program:

   $|[\text{CON}; \textbf{constr}\ \alpha; \textbf{inv}\ \beta; \textbf{pre}\ \varphi; \textbf{post}\ \psi]|$

   Then

   $$\begin{array}{lll} \text{wp}(\mathcal{P}, Q) & \triangleq \alpha \wedge \beta \wedge \varphi & \text{if } \alpha \wedge \beta \wedge \psi \Rightarrow Q \\ & \triangleq \textbf{false} & \text{otherwise.} \end{array}$$

Weakest precondition with constraints. If C and Q are formulas and $\mathcal{F}$ is an algorithm, then

wpc($\mathcal{F}/C$, Q) is the weakest formula establishing the constraint C and the postcondition Q. In other words, to show $\{P\}$ $\mathcal{F}/C$ $\{Q\}$, it is sufficient to show $P \Rightarrow$ wpc($\mathcal{F}/C$, Q).

1. Assignment: wpc(x: = E/C, Q) $\triangleq$ $C \Rightarrow Q(x \leftarrow E) \wedge C(x \leftarrow E) \wedge$ Def(E)
2. Composition: wpc($\mathcal{F}$; $\mathcal{G}/C$, Q) $\triangleq$ wpc($\mathcal{F}/C$; wpc($\mathcal{G}/C$, Q))
3. Simple IF: Let $\mathcal{P}$ be the following program:

   if $\gamma \rightarrow \mathcal{F}$
   $\square$ $\neg\gamma \rightarrow \mathcal{G}$
   fi

   then

   wpc($\mathcal{P}/C$, Q) $\triangleq$ $C \Rightarrow$ Def(γ) $\wedge$
   ($\gamma \Rightarrow$ wpc($\mathcal{F}/C$, Q)) $\wedge$ ($\neg\gamma \Rightarrow$ wpc($\mathcal{G}/C$, Q))

4. Simple verifiable loop: Let $\mathcal{P}$ be the following program:

   do $\gamma \rightarrow$
   invariant φ
   bound function E
   $\mathcal{F}$
   od

   We have to redefine the properties of the invariant φ in relation to C:

   $\{\varphi\}\mathcal{F}/C\{\varphi\} \wedge (\varphi \wedge C \Rightarrow$ Def(γ)), i.e., φ is an invariant of $\mathcal{F}$ with constraint C.

   Then

   wpc($\mathcal{P}/C$, Q) $\triangleq$ φ if $\varphi \wedge C \wedge \neg\gamma \Rightarrow Q$
   $\triangleq$ **false** otherwise.

5. Pseudocode instruction: Let α, β, φ, ψ be logical assertions and CON be an optional clause of the form **con** V, where V is a possible empty list of variables from the overall data structure, and let $\mathcal{P}$ be the following program:

   |[CON; **constr** α; **inv** β; **pre** φ; **post** ψ]|

   Then

   wpc($\mathcal{P}/C$, Q) $\triangleq$ $\alpha \wedge \beta \wedge \varphi$ if $(\alpha \wedge \beta \wedge \psi \Rightarrow Q) \wedge (\alpha \Rightarrow C)$
   $\triangleq$ **false** otherwise.

Appendix C: Proving correctness of star recursion

Preliminaries. "$\triangleq$" means "equal by definition," "$\Leftrightarrow$" means "if and only if," "$A \triangleq B \triangleq C$" is treated as "$A \triangleq B$ and $B \triangleq C$," and finally, "$A \Leftrightarrow B \Leftrightarrow C$" is treated as "$A \Leftrightarrow B$ and $B \Leftrightarrow C$." Each of "$\triangleq$" or "$\Leftrightarrow$" has a lower binding power than "=".

Since the constraint does not have occurrences of the variable identifiers, we may use the predicate transformer wp (see Appendix B).

The loop is well-defined in regard to its invariant and its bound function. Let us denote the constraint as C, the above loop as $\mathcal{L}$, its body as $\mathcal{F}$, the guard as γ, the invariant as φ, and its bound function as B. Note that in this case γ, φ, and B are everywhere defined and thus we may omit Def(γ), Def(φ), and Def(B) from the consideration. So, in order to show that φ and B are correctly defined, we must show

$(C \wedge \varphi \wedge \gamma \Rightarrow \text{wp}(\mathcal{F}, \varphi)) \wedge (C \wedge \varphi \Rightarrow \text{B} \geq 0) \wedge$
$(C \wedge \varphi \wedge \gamma\ \text{B} = \text{X} \Rightarrow \text{wp}(\mathcal{F}, \text{B} < \text{X}))$.

Note that $C \wedge \varphi \Rightarrow \text{B} \geq 0$ is trivially true since B $\triangleq$ BOUND(y) and BOUND is declared to be a total function returning nonnegative integers. Computing wp($\mathcal{F}$, φ):

wp($\mathcal{F}$, φ) $\triangleq$ wp(r := *head*(y) $\star$ r; y: = *tail*(y), φ)
$\triangleq$ wp(r: = *head*(y) $\star$ r, wp(y: = *tail*(y), φ))

wp(y: = *tail*(y), φ) $\triangleq$ $\varphi(y \leftarrow$ *tail*(y)) $\wedge$
Def(*tail*(y)) $\Leftrightarrow$ G(*yzero*) = G(*tail*(y)) $\star$ r

/* We have omitted Def(*tail*(y)) since *tail*(y) explicitly occurs in G(*yzero*) = G(*tail*(y)) $\star$ r. Subsequently, we will do such simplifications without comments. */

wp(r: = *head*(y) $\star$ r, G(*yzero*) = G(*tail*(y)) $\star$ r)
$\Leftrightarrow$ G(*yzero*) = G(*tail*(y)) $\star$ (*head*(y) $\star$ r)

Thus wp($\mathcal{F}$, φ) $\Leftrightarrow$ G(*yzero*) = G(*tail*(y)) $\star$ (*head*(y) $\star$ r)

Showing $C \wedge \varphi \wedge \gamma \Rightarrow$ G(*yzero*) = G(*tail*(y)) $\star$ (*head*(y) $\star$ r):

Assume $C \wedge \varphi \wedge \gamma$ to be true in some algorithm state. We must show that wp($\mathcal{F}$, φ) holds in the

same state. Since $\gamma \triangleq \neg low\_bound(y)$, by C both $tail(y)$ and $head(y)$ are defined. Since, by C, $\star$ is associative, get $G(tail(y)) \star (head(y) \star r) = (G(tail(y)) \star head(y)) \star r$. Since we have assumed that $\neg low\_bound(y)$ holds, the star recursion formula gives $G(y) = G(tail(y)) \star head(y)$. Replacing $(G(tail(y)) \star head(y))$ by $G(y)$ in the right side of the equation to be proved, get $G(yzero) = G(y) \star r$. The latter equation holds since φ is assumed to hold and since $\varphi \triangleq G(yzero) = G(y) \star r$. Done.

Showing that $C \wedge \varphi \wedge \gamma \wedge B = X \Rightarrow wp(\mathcal{F}, B < X)$ is just as straightforward. So the loop is correctly defined.

Correctness of the algorithm. By the rule for wp on compositions, proving the algorithm correct is equivalent to showing $C \Rightarrow wp(y := yzero; r := idstar, wp(\mathcal{L}; r := base(y) \star r, r = G(yzero)))$. Let us find $wp(\mathcal{L}; r := base(y) \star r, r = G(yzero))$:

$wp(\mathcal{L}; r := base(y) \star r, r = G(yzero)) \triangleq wp(\mathcal{L}, wp(r := base(y) \star r, r = G(yzero))) \triangleq wp(\mathcal{L}, base(y) \star r = G(yzero))$

We must find $wp(\mathcal{L}, base(y) \star r = G(yzero))$ in two steps:

1. Showing $C \wedge \varphi \wedge \neg\gamma \Rightarrow base(y) \star r = G(yzero)$:

   Since $\neg\gamma \Leftrightarrow low\_bound(y)$, C implies that $base(y)$ is defined and that $G(y) = base(y)$. Thus it is sufficient to show $G(y) \star r = G(yzero)$. This follows from $\varphi \triangleq G(yzero) = G(y) \star r$.

2. $wp(\mathcal{L}, base(y) \star r = G(yzero)) \triangleq \varphi \triangleq G(yzero) = G(y) \star r$.

Therefore, to show correctness of the whole algorithm we must show

$C \Rightarrow wp(y := yzero; r := idstar, \varphi)$.

This is straightforward.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of the U.S. Department of Defense.

Cited references and notes

1. E. W. Dijkstra, *A Discipline of Programming,* Prentice-Hall, Inc., Englewood Cliffs, NJ (1976).
2. D. Gries, *The Science of Programming*, Springer-Verlag, Inc., New York (1981).
3. A. Kaldewaij, *Programming: The Derivation of Algorithms*, Prenctice-Hall, Inc., Englewood Cliffs, NJ (1990).
4. E. Cohen, *Programming in the 90s: An Introduction to the Calculation of Programs*, Springer-Verlag, Inc., New York (1990).
5. V. Yakhnis, J. Farrell, and S. Shultz, "A Case Study in Deriving Programs Using Generic Algorithms," to be submitted for publication.
6. In this context "rigorous" means "mathematically precise" and thus pertains to semantics. In contrast, "informal" pertains only to syntax. Thus "informal rigorous" is not self-contradictory.
7. E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics*, Springer-Verlag, Inc., New York (1990).
8. J. M. Spivey, *The Z-notation: A Reference Manual*, Prentice-Hall International, Englewood Cliffs, NJ (1989).
9. J. M. Spivey, *Understanding Z: A Specification Language and Its Formal Semantics*, Cambridge University Press, Cambridge, England (1988).
10. D. C. Ince, *An Introduction to Discrete Mathematics and Formal System Specification*, Oxford University Press, Oxford (1988).
11. J. Woodcock and M. Loomes, *Software Engineering Mathematics*, Pitman (1988).
12. C. B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall International, Englewood Cliffs, NJ (1990).
13. R. M. Burstall and J. A. Goguen, "Putting Theories Together to Make Specifications," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (1977), pp. 1045–1058.
14. J. V. Guttag and J. J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, Inc., New York (1993).
15. C. Morgan, *Programming from Specifications*, Oxford University Press, Oxford (1991).
16. Usually in the literature the term *variable* is used instead. However, it is just as usual to assign to the term variable a more narrow meaning (i.e., something that may be changed). This may cause confusion since some identifiers may not be changed.
17. S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs," *Acta Informatica*, No. 6, 319–340 (1976).
18. This division may be artificial, but discussing this is beyond the scope of this paper.
19. M. Wirsing, "Algebraic Specification," *Handbook of Theoretical Computer Science*, Elsevier Science Publishers B.V., Amsterdam (1990), pp. 675–788.
20. D. Harel, *First-Order Dynamic Logic*, Springer-Verlag, Inc., New York (1979).
21. J. Loeckx and K. Sieber, *The Foundations of Program Verification*, John Wiley & Sons, Inc., New York (1987).
22. We may use pseudocode instructions to override constraints and to specify instructions that are "atomic" in respect to the overall constraint, but such discussion is beyond the scope of the present paper.
23. E. W. Dijkstra and W. H. J. Feijen, *A Method of Programming*, Addison-Wesley Publishing Co., Reading, MA (1988).
24. J. Bohorquez and R. Cardoso, "Problem Solving Strategies for the Derivation of Programs," to appear in *Logical*

Methods, a collection of papers honoring A. Nerode's 60th birthday.
25. R. B. Kieburtz and J. Shultis, "Transformations of FP Program Schemes" *Proceedings of the Conference on Functional Programming and Architecture* (1981), pp. 41–48.
26. V. Yakhnis, J. Farrell, and S. Shultz, "A Library of Generic Algorithms in Dijkstra-Gries Notation," to appear.
27. T. Clement, "The Role of Data Reification in Program Refinement: Origin, Synthesis and Appraisal," *The Computer Journal* **35**, No. 5, 441–450 (October 1992).
28. J. C. P. Woodcock, "The Rudiments of Algorithm Refinement," *The Computer Journal* **35**, No. 5, 431–440 (October 1992).
29. J. R. Shoenfield, *Mathematical Logic*, Addison-Wesley Publishing Co., Reading, MA (1967).
30. R. Breu, *Algebraic Specification Techniques in Object Oriented Programming Environments*, Springer-Verlag, Inc., New York (1991).
31. J. V. Tucker and J. I. Zucker, *Program Correctness over Abstract Data Types with Error-State Semantics*, North-Holland, Amsterdam (1988).

General references

K. R. Apt, "Ten Years of Hoare's Logic, a Survey," *ACM Transactions on Programming Languages and Systems* **3**, 431–483 (1981).

K. R. Apt and E. R. Olderog, *Verification of Sequential and Concurrent Programs*, Springer-Verlag, Inc., New York (1991).

R. C. Backhouse, *Program Construction and Verification*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1986).

M. Dyer, *The Cleanroom Approach to Quality Software Development*, John Wiley & Sons, Inc., New York (1992).

D. Gumb, *Programming Logics*, John Wiley & Sons, Inc., New York (1989).

C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM* **12**, No. 10, 576–580, 583 (1969).

C. A. R. Hoare, "An Axiomatic Approach to Computer Programming," *Essays in Computer Science*, C. A. R. Hoare and C. B. Jones, Editors, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989).

R. C. Linger, H. D. Mills, and B. L. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley Publishing Co., Reading, MA (1979).

H. D. Mills, R. C. Linger, and A. R. Hevner, *Principles of Information Systems Analysis and Design*, Academic Press, Inc., New York (1986).

C. Reynolds, *The Craft of Programming*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1981).

Accepted for publication September 20, 1993.

Vladimir R. Yakhnis *IBM Large Scale Computing Division, 1701 North Street, Endicott, New York 13760.* Dr. Yakhnis is an advisory programmer. He received a Diploma in mathematics from Moscow State University, Moscow, Russia, in 1975. He worked as a computer programmer in Moscow and in Houston, Texas. Dr. Yakhnis received a Ph.D. in mathematics and an M.S. in computer science from Cornell University in 1990. He worked as a Postdoctoral Fellow at the Mathematical Sciences Institute, Cornell University, until 1991. His research was in program correctness for concurrent and sequential programs, winning strategies for two-person games, finite automata, and recursion theory. He joined IBM in 1991 in Endicott.

Joel A. Farrell *IBM Large Scale Computing Division, 1701 North Street, Endicott, New York 13760.* Mr. Farrell is a senior programmer. He joined IBM in Endicott in 1981 in VM Development where he has worked on such areas as I/O and program management. Most recently he has been involved in parallel and distributed computing and in formal methods for software development. Mr. Farrell received his B.S. degree in computer science from Kansas State University in 1980 and his M.S. degree in 1985 from Syracuse University, also in computer science. In 1992 he received an IBM Outstanding Innovation Award for his work on parallel processing in CMS, and an IBM First Level Invention Achievement Award. He is a member of the Association for Computing Machinery.

Steven S. Shultz *IBM Large Scale Computing Division, 1701 North Street, Endicott, New York 13760.* Mr. Shultz is an advisory programmer. He joined IBM in Endicott in 1981 in VM Development where he has worked on such areas as I/O and virtual storage management. Most recently he has been involved in parallel and distributed computing and in formal methods for software development. Mr. Shultz received his B.S. degree in computer and information science from Ohio State University in 1981. In 1992 he received an Outstanding Innovation Award for his work on parallel processing in CMS and an IBM First Level Invention Achievement Award.

Reprint Order No. G321-5537.