Formal Verification of a Merge-Sort Program with Static Semantics

Paul E. Black
National Institute of Standards and Technology (NIST)
paul.black@nist.gov

George Becker*
Department of Computer Science
University at Albany - SUNY
becker@cs.albany.edu

Neil V. Murray*
Department of Computer Science
University at Albany - SUNY
nvm@cs.albany.edu

Abstract

Proving correctness of programs is a desirable, but not yet practically solved, problem. Conventional verification techniques do not scale to larger "real-life" programs. In order to overcome problems with the semantics of conventional languages, researchers define subsets of these languages for verification. We take this approach one step further; rather than defining a subset, we apply a new theory based on relaxed single-threading to achieve completely static semantics (i.e. referential transparency) of programs. Our programs with static semantics have complete control over aliasing, the correct order dependencies are enforced, and uninitialized variables are prevented.

Due to the static semantics of these programs, first-order logic can be used directly to verify them. Using Merge-Sort as an example, we demonstrate that first-order logic proofs of programs with static semantics are fully composable and thus scale freely to larger programs. We also report on our work towards a fully formal, machine-checked proof. This work is one more step towards facilitating the design of more easily verified programs.

1 Introduction

The desirability of proving programs correct with respect to their specifications has been suggested many times [Dij, Flo, Gri, Ho1, Ho2, Man, McM, BW]. However, program proving appears to be a very costly and difficult exercise [MLP]. A variety of principles of program proving, or verification, have been applied successfully to some small programs [Ho2]. The applicability of program verification to larger, "real-life" programs has been disputed [MLP].

One of the main difficulties in applying the known techniques of program verification is the combinatorial complexity of large proofs. Due to the lack of composability of these techniques, they do not scale to larger, "real-life" programs. Using first-order logic instead facilitates program verification. This is much easier for programs with static semantics.

Formal verification can become practical and successful in having a significant impact on practice if verification is more tractable, more productive, and cost-effective. Due to problems with semantics of conventional programming languages, researchers in formal verification define subsets of those languages. For example, the Penelope system [EHH] developed by Odyssey Research Associates, Inc. uses a subset of Ada. Their intent is to verify, by static semantic checks, that programs are free from improper aliasing, undisciplined side effects, incorrect order dependencies, and uses of uninitialized variables.

We take such approaches one step further. Rather than defining a subset of a conventional language, we adopt the single-assignment style to achieve static semantics (i.e., referential transparency) of our programs. Our use of single-assignment is in the declarative sense of functional programming, that is as a definition, rather than in the operational sense of imperative programming as a side effect. Programs with these static semantics have complete control over aliasing, correct order dependencies are enforced, and uninitialized variables are prevented. Most importantly, due to the static semantics, first-order logic can be used directly to verify programs, thus avoiding difficult verification techniques based on temporal first-order logics [MP] or Floyd-Hoare axiomatization [Flo, Ho1]. Since we are able to use a few predefined higher-order functions rather than recursion in our programs, we can apply mathematical induction. Due to the full compositionality of programs with static semantics, our approach can freely scale to larger programs.

Section 2 discusses theoretical foundations of our work. Section 3 presents our Merge-Sort program consisting of four function definitions. Section 4 then gives the proof of program correctness. We report our progress toward a machine proof in Section 5 and discuss extensions, implications, and future work in

* Supported in part by NSF Grant CCR-9404338.

Section 6. Appendix A lists standard operators and functions we use, and Appendix B formally defines program primitives.

2 Theoretical Basis

We developed an underlying theory [BM] that makes it possible to achieve a completely static semantics. Our approach is motivated by *single-threading* (i.e. using a value only once) of aggregate data (e.g., arrays, lists, trees, tuples). Linear LISP [Bak] makes single-threading explicit at the programmer's level. This approach *restricts* a functional programming language so that *only singly-threaded programs can be expressed.* However, rigid enforcement of single-threading is a severe constraint on the expressiveness of a programming language, and thus appears to be self-defeating.

We relax single-threading to restore expressiveness [BM] while still being able to benefit from the limited single-threading that remains. The central idea is to divide *function input arguments* into *two categories* called *inert inputs* and *active inputs.* This distinction is crucial because aggregate data arguments are passed by reference for efficiency. *Inert* inputs are used for (aggregate) data not subject to updating; e.g., when evaluating the length of a list or computing the sum of a vector of numbers. Such operations require no copying of data. *Active* inputs are used for aggregate data subject to updating in a given operation.

Our relaxed single-threading has two aspects. First, our *structured function composition* defines legal uses of inert and active inputs when composing functions. Second, our *memory axioms* stipulate how predefined primitive functions must be implemented. We have shown [BM] that evaluation of any structured function composition of primitive functions is free of aliasing, i.e. single-threading is preserved. Thus static semantics, also known as *referential transparency*, is achieved despite in-place updating of active input data.

3 Merge-Sort Program

We illustrate our approach on a Merge-Sort program for lists of (real) numbers. The program applies to any type of element as long as elements are comparable.

Inert and active inputs can be distinguished by either a separator (the bar character in [BM]) or by suitable annotations (keywords). Since annotations of inert and active inputs do not change the semantics of the program, they have no effect on the program proof. Since this paper concentrates on verification, we omit these annotations in our Merge-Sort program. The annotations are only needed for compilers to determine legal programs and to translate programs correctly. The various "utility" functions used in the program are formally defined in the Appendix B.

The program's top-level function, `merge_sort`, first converts the input data into a list of singleton lists. Since each singleton list has only one element, they are trivially sorted. The core of `merge_sort` is repeatedly applying the function `msort_step`. The function `msort_step` sorts and merges the list of many singleton lists into a list of one sorted list containing all the input elements. The function `flatten` gives the result as a simple list.

Function `msort_step` splits its list-of-lists argument into two halves, then merges each list from the first half with a (corresponding) list from the second half. Successive applications of `msort_step` combine "inner lists," keeping each inner list sorted, which shortens the "outer list." Eventually the outer list has just one list: a sorted inner list with all the original elements.

Function `ord_merge` just merges its two list arguments. Since we keep the two argument lists sorted, the resulting list is sorted as well. Function `merge_step` is an auxiliary function that performs one step of the ordered list merge by choosing the smaller element of two lists. The chosen element is then transferred to the list holding the intermediate result of merging.

```
function merge_sort(list(real) A):
    int B = ceiling(log2(max(size(A),1))),
    list(list(real)) C = explode_list(A),
    list(list(real)) Y = iter(B,msort_step,C),
    list(real) Z = flatten(Y),
    result (Z).

function msort_step(list(list(real)) C):
    (list(list(real)) D, list(list(real)) E)
                         = split_list(C),
    list(list(real)) X
                  = map2_fill(ord_merge,D,E),
    result (X).

function ord_merge(list(real) D,
                   list(real) E):
    int F = size(D) + size(E),
    (list(real) U, list(real) V, list(real) W)
          = iter(F,merge_step,D,E,empty_list),
    result (W).

function merge_step(list(real) D,
                  list(real) E, list(real) G):
    bool H = size(D)==0 or first_greater(D,E),
    (list(real) P, list(real) Q)
                         = swap_if(H,D,E),
    (list(real) R, list(real) S)
                       = headlist_tail(P),
    list(real) T = concat(G,R),
    result (Q,S,T).
```

4 Proof of Program Correctness

We prove the correctness of this program by proving four theorems. Each theorem specifies the properties of one of the functions in the program. Our proof is a brief sketch with many details left out because of length. Appendix B defines two predicates (`sorted`, `permut`) and three functions (`concat`, `concat3`, `flatten`) that we also use in the proof. The term $\mathrm{permut}(A, B)$ means that A is a permutation of B. Appendix B also defines sequence comparison (e.g., list comparison) which we denote by $\preceq$.

4.1 Function merge_step

First, we give the necessary precondition (1) for this function:

$$\left.\begin{array}{c} \mathrm{sorted}(D) \wedge \mathrm{sorted}(E) \wedge \mathrm{sorted}(G) \\ \wedge \\ (G \preceq D) \wedge (G \preceq E) \end{array}\right\} \quad (1)$$

Now, we give the desired postcondition (2) for the results of applying function `merge_step`:

$$\left.\begin{array}{c} \mathrm{sorted}(Q) \wedge \mathrm{sorted}(S) \wedge \mathrm{sorted}(T) \\ \wedge \\ (T \preceq Q) \wedge (T \preceq S) \\ \wedge \\ \mathrm{permut}(\mathrm{concat3}(Q, S, T), \mathrm{concat3}(D, E, G)) \\ \wedge \\ (\mathrm{size}(Q) + \mathrm{size}(S) = \\ \qquad \max(0, \mathrm{size}(D) + \mathrm{size}(E) - 1)) \end{array}\right\} \quad (2)$$

Theorem 1 *If function* merge_step *is applied to arguments satisfying precondition (1), then the function results satisfy postcondition (2).*

Proof: Assume that the precondition (1) holds. The value of variable H is true, if and only if either list D is empty or lists D and E are both non-empty and the first element of list D is greater than the first element of list E. Otherwise, the value of H is false. For all of these cases, the following condition (3) holds:

$$\left.\begin{array}{c} \mathrm{sorted}(P) \wedge \mathrm{sorted}(Q) \\ \wedge \\ (\mathrm{size}(P) = 0 \vee \mathrm{size}(Q) = 0 \\ \qquad \vee \min(P) \leq \min(Q)) \\ \wedge \\ \mathrm{permut}(\mathrm{concat}(P, Q), \mathrm{concat}(D, E)) \end{array}\right\} \quad (3)$$

We note that P and Q are sorted. If P and Q are also non-empty, then the first element of P is equal to $\min(P)$, and the first element of Q is equal to $\min(Q)$. Since the first element of P is less than or equal to the first element of Q, $\min(P) \leq \min(Q)$. We use the condition $\min(P) \leq \min(Q)$ since it is better for carrying out the proof.

From the precondition (1) and from properties of function `headlist_tail`, we obtain the following formula (4) for lists R and Q:

$$\left.\begin{array}{c} (\mathrm{size}(R) = \min(1, \mathrm{size}(P))) \\ \wedge \\ \mathrm{sorted}(R) \wedge \mathrm{sorted}(S) \wedge (R \preceq Q) \wedge (R \preceq S) \\ \wedge \\ \mathrm{permut}(\mathrm{concat}(R, S), P) \end{array}\right\} \quad (4)$$

From (1), (3), (4) and properties of function `concat`, we get the formula (5) for list T:

$$\mathrm{sorted}(T) \wedge \mathrm{permut}(T, \mathrm{concat}(G, R)) \quad (5)$$

The postcondition (2) for function `merge_step` can be obtained from (1), (3), (4) and (5).

4.2 Function ord_merge

We formulate the necessary precondition (6) for this function:

$$\mathrm{sorted}(D) \wedge \mathrm{sorted}(E) \quad (6)$$

We give the desired postcondition (7) for the results of applying function `ord_merge`:

$$\mathrm{sorted}(W) \wedge \mathrm{permut}(W, \mathrm{concat}(D, E)) \quad (7)$$

Theorem 2 *If function* ord_merge *is applied to arguments satisfying precondition (6), then the function results satisfy postcondition (7).*

Proof: Assume that the precondition (6) holds. We proceed by induction on F, the number of applications of function `merge_step`. We define predicate $\mathrm{Pr1}(F)$ as follows:

$$\begin{aligned} \mathrm{Pr1}(F) \;=\; & \mathrm{sorted}(U) \wedge \mathrm{sorted}(V) \wedge \mathrm{sorted}(W) \wedge \\ & (W \preceq U) \wedge (W \preceq V) \wedge \\ & \mathrm{permut}(\mathrm{concat3}(U, V, W), \mathrm{concat}(D, E)) \wedge \\ & (\mathrm{size}(U) + \mathrm{size}(V) \\ & \qquad = \max(0, \mathrm{size}(D) + \mathrm{size}(E) - F)) \end{aligned}$$

Base Case: Let $F = 0$. Then $U = D$, $V = E$, and $W = \mathrm{empty\_list}$, and so $\mathrm{Pr1}(0)$ holds.
Ind. Step: Assume that $\mathrm{Pr1}(F)$ holds for some non-negative integer. We get for $F + 1$:

$$\begin{aligned} & \mathrm{iter}(F+1, \mathrm{merge\_step}, D, E, \mathrm{empty\_list}) \\ & \quad = \mathrm{merge\_step}(\mathrm{iter}(F, \mathrm{merge\_step}, D, E, \mathrm{empty\_list})) \end{aligned}$$

From the induction hypothesis $\mathrm{Pr1}(F)$, the precondition (6) for the *(F+1)-th* application of function `merge_step` is satisfied. Thus we can apply Theorem 1. The permutation clause of $\mathrm{Pr1}(F + 1)$ results

from $\mathrm{Pr1}(F)$, Theorem 1 and transitivity of permutation. As each application of function `merge_step` decreases $\mathrm{size}(U)+\mathrm{size}(V)$ by one (unless already zero), the last clause in $\mathrm{Pr1}(F+1)$ also holds. □
As we have $F = \mathrm{size}(D) + \mathrm{size}(E)$, we obtain:

$$\left.\begin{array}{rcl} \mathrm{size}(U) & + & \mathrm{size}(V) \\ & = & \max(0, \mathrm{size}(D) + \mathrm{size}(E) - F) \\ & = & \max(0, F - F) = 0 \end{array}\right\} \quad (8)$$

From predicate $\mathrm{Pr1}(F)$ and (8), we get:

$$\mathrm{permut}(W, \mathrm{concat}(D, E)) \quad (9)$$

The postcondition (7) for function `ord_merge` follows immediately from predicate $\mathrm{Pr1}(F)$ (i.e. $\mathrm{sorted}(W)$) and (9).

4.3 Function msort_step

We formulate the necessary precondition (10) for this function:

$$\forall_{1\leq i\leq \mathrm{size}(C)}\ i : \mathrm{sorted}(c_i) \quad (10)$$

We give the necessary postcondition (11) for the results of applying function `msort_step`:

$$\left.\begin{array}{c} (\mathrm{size}(X) = \lceil \mathrm{size}(C)/2 \rceil) \\ \wedge \\ (\forall_{1\leq i\leq \mathrm{size}(X)}\ i : \mathrm{sorted}(x_i)) \\ \wedge \\ \mathrm{permut}(\mathrm{flatten}(X), \mathrm{flatten}(C)) \end{array}\right\} \quad (11)$$

Theorem 3 *If function* msort_step *is applied to arguments satisfying precondition (10), then the function results satisfy postcondition (11).*

Proof: Assume that the precondition (10) holds. From this precondition and properties of function `split_list`, we get:

$$\left.\begin{array}{c} (\mathrm{size}(D) = \lceil \mathrm{size}(C)/2 \rceil) \\ \wedge \\ (\mathrm{size}(E) = \lfloor \mathrm{size}(C)/2 \rfloor) \\ \wedge \\ (\forall_{1\leq i\leq \mathrm{size}(D)}\ i : \mathrm{sorted}(d_i)) \\ \wedge \\ (\forall_{1\leq j\leq \mathrm{size}(E)}\ j : \mathrm{sorted}(e_j)) \\ \wedge \\ \mathrm{permut}(\mathrm{flatten}(C), \\ \qquad \mathrm{concat}(\mathrm{flatten}(D), \mathrm{flatten}(E)) \end{array}\right\} \quad (12)$$

From (12), properties of function `map2_fill` and Theorem 2, we obtain:

$$\left.\begin{array}{c} (\mathrm{size}(X) = \lceil \mathrm{size}(C)/2 \rceil) \\ \wedge \\ (\forall_{1\leq i< \mathrm{size}(X)}\ i : \mathrm{sorted}(x_i)) \\ \wedge \\ (\forall_{1\leq j< \mathrm{size}(X)}\ j : \mathrm{permut}(x_j, \mathrm{concat}(d_j, e_j)) \\ \wedge \\ \text{if } \mathrm{odd}(\mathrm{size}(C)) \\ \quad \text{then } \mathrm{permut}(x_{\mathrm{size}(X)}, d_{\mathrm{size}(D)}) \\ \quad \text{else } \mathrm{permut}(x_{\mathrm{size}(X)}, \\ \qquad \mathrm{concat}(d_{\mathrm{size}(D)}, e_{\mathrm{size}(E)})) \end{array}\right\} \quad (13)$$

The $\mathrm{size}(C)$ is either odd or even. In both cases, the postcondition (11) follows from (13) and (12).

4.4 Function merge_sort

Precondition: None
We formulate the desired postcondition (14) for the results of applying function `merge_sort`:

$$\mathrm{permut}(Z, A) \wedge \mathrm{sorted}(Z) \quad (14)$$

Theorem 4 *Applying the function* merge_sort *satisfies the postcondition (14).*

Proof: From properties of function `explode_list`, we obtain:

$$\left.\begin{array}{c} (\mathrm{size}(C) = \mathrm{size}(A)) \\ \wedge \\ (\forall_{1\leq i\leq \mathrm{size}(C)}\ i : \mathrm{sorted}(c_i)) \\ \wedge \\ \mathrm{permut}(\mathrm{flatten}(C), A) \end{array}\right\} \quad (15)$$

We continue by induction on B, the number of applications of function `msort_step`. We define $\mathrm{Pr2}(B)$ as follows:

$$\begin{array}{rcl} \mathrm{Pr2}(B) & = & (\forall_{1\leq j\leq \mathrm{size}(Y)}\ j : \mathrm{sorted}(y_j)) \wedge \\ & & \mathrm{permut}(\mathrm{flatten}(Y), \mathrm{flatten}(C)) \wedge \\ & & (\mathrm{size}(Y) \leq \max(1, 2^{\lceil \log_2(\max(\mathrm{size}(A),1)\rceil - B})) \end{array}$$

Base Case: Let $B = 0$. Then $Y = A$, and so $\mathrm{Pr2}(0)$ holds.
Ind. Step: Assume $\mathrm{Pr2}(B)$ holds for some non-negative integer. We get for $B+1$:

$$\begin{array}{l} \mathrm{iter}(B+1, \mathrm{msort\_step}, C) \\ \qquad = \mathrm{msort\_step}(\mathrm{iter}(B, \mathrm{msort\_step}, C)) \end{array}$$

From the induction hypothesis $\mathrm{Pr2}(B)$, the precondition (10) for the *(B+1)-th* application of function `msort_step` is satisfied. Thus we can apply Theorem 3. The permutation clause of $\mathrm{Pr2}(B+1)$ results from $\mathrm{Pr2}(B)$, Theorem 3 and transitivity of permutation. As each application of function `msort_step` decreases the upper bound of $\log_2(\max(\text{size}(Y), 1))$ by one (unless $\text{size}(Y) \leq 1$ already), the last clause of $\text{Pr2}(B+1)$ also holds. □

As we have $B = \lceil \log_2(\max(\text{size}(A), 1)) \rceil$, $\text{size}(Y) \leq \max(1, 2**0) = 1$. Since the elements of Y are sorted, $\text{size}(Y) \leq 1$ means that sorted(flatten(Y)) is true.

We further obtain from $\text{Pr2}(B)$ and (15):

$$\left.\begin{array}{c} \text{permut}(\text{flatten}(Y), \text{flatten}(C)) \\ \wedge \\ \text{permut}(\text{flatten}(C), A) \end{array}\right\} \qquad (16)$$

From (16) and from the facts that sorted(flatten(Y)) is trivially true and $Z = \text{flatten}(Y)$, we obtain the postcondition (14) for the function `merge_sort`. Naturally, $\text{size}(Z) = \text{size}(A)$ is also true.

5 Machine Proofs Efforts

To be practical for more and larger programs, proofs of correctness must be at least partially automated. To this end we are working on fully formal, machine-checked proofs in HOL [GM] and PVS [ORS]. To prove correctness we need three formally defined components: a specification, an implementation model, and rules of inference [BHJ+].

We formalize the preconditions and postconditions from Section 4 as the specifications. We translate the program from Section 3 into functional descriptions in the built-in first-order logics as the implementation. (This is a "shallow" embedding.) Our models refer to sorting natural numbers (non-negative integers), instead of real numbers, since they are a "native type" and thus a little easier to reason about. We use the standard, built-in rules of logic, math, and function composition for rules of inference.

With the specification and a model of the implementation described formally, the "theorem provers" check that each step of the proof is correct. Although HOL and PVS prove some of the simplest steps automatically, the fully formal proof is still many times more complex than the proofs in this paper. For instance, obvious and technical lemmas such as $(\text{sorted}(G) \wedge \forall y \in G : x \geq y) \Rightarrow \text{sorted}(\text{append}(G, x))$ must be proved in great detail.

Our work so far in formalizing and proving the specifications and program found some typographical errors which had escaped earlier manual checking.

6 Discussion

The purpose of our Merge-Sort program is to provide a good illustration of our proof method on a familiar algorithm. Even though actual applications of the Merge-Sort algorithm tend to operate on external data (e.g., magnetic tapes), for purposes of illustration, this Merge-Sort program operates on internal data. We chose to represent sequences as lists, since lists are more natural and better suited for expression of the Merge-Sort algorithm. Unlike a Merge-Sort algorithm coded in conventional sequential languages (e.g., Pascal, Modula-2, Ada, C/C++, etc.), our Merge-Sort is, in principle, a parallel program. In particular, the higher order function `map2_fill` can be viewed as a data-parallel construct.

Our Merge-Sort program uses higher order functions and is nonrecursive. Its call-graph is a (rooted) tree. We carry out its proof by simple induction. The "loop invariants" of conventional proofs are captured by our induction predicates. The proof proceeds in a "bottom-up" fashion from leaves of the call-graph tree towards its root. Naturally, when proving that a function corresponding to an interior node of the call-graph tree, we use intermediate proof results from children of the interior node. Other than that, however, proofs of individual functions making up a program are independent of each other. Consequently, our proofs are composable and thus can scale to larger programs.

Conventional proof methods show *total correctness* by establishing *partial correctness* of a program and separately proving *program termination*. Our method shows correctness *entirely* by proving that an input-output transformation of a program satisfies the given specification. If the input-output transformation is a total function, the termination of a program follows. In the case of our Merge-Sort program, there is no possibility of non-termination since no construct in our program admits infinite iteration. Formally speaking, the input-output transformation of Merge-Sort is *primitive recursive*, and thus total.

The classical proof of *Find*, which performs partial sorting, is described in Sections 2, 3, and 4 of [Ho2], and includes the following reservation in Section 5:

> In the proof of Find, one very important aspect of correctness has not been treated, namely, that the program merely rearranges the elements of the array A, without changing their values.

The proof of our Merge-Sort program shows directly that the output is a permutation (i.e. rearrangement) of the input. Currently, we are considering proving the program *Find* [Ho2] by our method. In the future, we wish to further demonstrate compositionality of our method. Since compositionality is a necessary condition for scalability and thus for the success of any verification technique, we plan to show scalability by verifying larger programs, including a simple text analyzer (histogram generator) and a maximum network flow program.

Acknowledgments

We express our sincere thanks to Richard E. Stearns and S. S. Ravi for helpful discussions on this paper. We

thank Trent N. Larson and William Majurski for their work on the machine proof.

References

[Bak] Henry G. Baker. A 'Linear Logic' Quicksort. In *ACM Sigplan Notices*, Feb 1994.

[BM] George Becker and Neil V. Murray. Efficient execution of programs with static semantics. In *ACM Sigplan Notices*, April 1995.

[BMS] George Becker, Neil V. Murray, and Richard E. Stearns. Refined single-threading for parallel functional programming. In *Languages, Compilers and Run-Time Systems for Scalable Computers* (Eds.: Boleslaw K. Szymanski and Balaram Sinharoy), Kluwer Academic Publishers, 1996.

[BHJ+] Paul E. Black, Kelly M. Hall, Michael D. Jones, Trent N. Larson, and Phillip J. Windley. A brief introduction to formal methods. In *Proc. of CICC '96*, pages 377–380, 1996.

[BW] Paul E. Black and Phillip J. Windley. Formal verification of secure programs in the presence of side effects. In *Proc. of HICSS-31*, vol. III, pages 327–334, 1998.

[Dij] Edsger W. Dijkstra. *A Discipline of Programming.* Englewood Cliffs/Prentice-Hall, 1976.

[EHH] Carl T. Eichenlaub, C. Douglas Harper and Geoffrey Hird. Using Penelope to Asses the Correctness of NASA Ada Software: A Demonstration of Formal Methods as a Counterpart to Testing. NASA Contractor Report 4509, May 1993.

[Flo] Robert W. Floyd. Assigning meanings to programs. In *Proc. of Symposium in Applied Mathematics*, pages 19–32, 1967.

[GM] Michael J. C. Gordon and Tom F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[Gri] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[Ho1] C. A. R. Hoare. An axiomatic basis for computer programming. In *Comm. of ACM*, **12** (10), pages 576–583, 1969.

[Ho2] C. A. R. Hoare. Proof of a program FIND. In *Comm. of ACM*, **14** (1), pages 39–45, 1971.

[Man] Zohar Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[MP] Zohar Manna and Amir Pnueli. Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag, 1992.

[McM] John McCarthy. Towards a mathematical science of computation. In *IFIP 62*, pages 21–29, Amsterdam. North-Holland, 1962

[MLP] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis Social processes and proofs of theorems and programs. In *Comm. of ACM*, **22** (5), pages 271–280, 1979.

[ORS] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: a prototype verification system. In *11th CADE*, vol. 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, 1992.

A Standard Operators and Functions

In our logical formulas and in the program, we use standard arithmetic, relational and logical operators $+$, $-$, $*$, $/$, $<$, $\leq$, $=$, $\neg$, $\wedge$, $\vee$. In the program, we use "`or`" for the logical operator $\vee$, as well as `==` for the relational operator $=$. We also use the functions `max`, `min`, `ceiling` ($\lceil x \rceil$), binary exponentiation (2^x) and binary logarithm ($\log_2$) with their standard meaning. In our logical formulas (but not in the program), the function `min` is not only a binary function but is extended in the usual way to non-empty sequence operands.

Except for the data being sorted, all other atomic data types are either boolean or integer. In fact, both the program and the proof require only natural numbers, not integers. Since we wish to avoid defining additional functions and complicating our logical formulas, these formulas do contain a few occurrences of arithmetic subexpressions that can evaluate to negative integers. Similarly, a few arithmetic subexpressions may evaluate to real values. However, these subexpressions occur only within integer expressions.

B Definitions of Primitive Functions and Predicates

We use the following functions, as well as the predicates `odd`, `sorted`, `permut`, and $\preceq$ (sequence comparison) in our logical formulas and/or in the program.

Definition 1 *Predicate* odd*:*
$\mathrm{odd}(n) = \neg(\exists m \in \mathcal{N}at : n = 2 * m)$

Let $Q = q_1, q_2, q_3, ..., q_k$ be a finite sequence (of, e.g., numbers).

Definition 2 *Size (length or number of elements) of a sequence Q:*
$$\mathrm{size}(Q) = \begin{cases} k & \text{if } Q \text{ is non-empty} \\ 0 & \text{otherwise} \end{cases}$$

Definition 3 *Prefix of length n of a sequence Q:*
$\text{prefix}(n, Q) = Q'$, *such that* $\text{size}(Q') = \min(n, \text{size}(Q))$ *and* $\forall_{1 \leq i \leq \text{size}(Q')}\ q'_i = q_i$

Definition 4 *Suffix of length n of a sequence Q:*
$\text{suffix}(n, Q) = Q'$, *such that* $\text{size}(Q') = \min(n, \text{size}(Q))$ *and* $\forall_{1 \leq i \leq \text{size}(Q')}\ q'_i = q_{i+\text{size}(Q)-\text{size}(Q')}$

Definition 5 *Tail of a sequence:*
$\text{tail}(Q) = \text{suffix}(\max(0, \text{size}(Q) - 1), Q)$

Definition 6 *Prepend an element q_p to a sequence Q:*
$\text{prepend}(q_p, Q) = Q'$, *such that* $\text{size}(Q') = \text{size}(Q) + 1$, $q'_1 = q_p$, *and* $Q = \text{tail}(Q')$

Definition 7 *Sorted Property (in a non-decreasing order) of a sequence Q:*
$\text{sorted}(Q) = \forall_{1 \leq i < \text{size}(Q)}\ i : q_i \leq q_{i+1}$

Definition 8 *Counting elements of sequence S having given value v:*
$\text{count}(v, S) = \sum_{1 \leq i \leq \text{size}(S)} (\text{if } s_i = v \text{ then } 1 \text{ else } 0)$

Definition 9 *Sequence comparison:*
Let S_1 and S_2 be sequences of comparable elements.
$S_1 \preceq S_2 = \forall s_1 \in S_1, s_2 \in S_2 : s_1 \leq s_2$

Definition 10 *Minimum element value in a sequence (e.g., a list):*

$$\min(Q) = \begin{cases} \min(q_1, \min(\text{tail}(Q))) & \text{if size}(Q) > 1 \\ q_1 & \text{if size}(Q) = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 11 *Comparison of first elements of two sequences R and S:*

$$\text{first\_greater}(Q, S) = (\text{size}(Q) > 0 \land \text{size}(S) > 0 \land q_1 > s_1)$$

Definition 12 *Conditional swapping of two arguments:*

$$\text{swap\_if}(Cond, x, y) = \begin{cases} y, x & \text{if } Cond \\ x, y & \text{otherwise} \end{cases}$$

Definition 13 *Concatenation of two sequences:*
$\text{concat}(Q, R) = S$
such that $\text{size}(S) = \text{size}(Q) + \text{size}(R)$,
$Q = \text{prefix}(\text{size}(Q), S)$, $R = \text{suffix}(\text{size}(R), S)$.

Definition 14 *Concatenation of three sequences:*
$\text{concat3}(Q_1, Q_2, Q_3) = \text{concat}(Q_1, \text{concat}(Q_2, Q_3))$

Definition 15 *Permutation as a binary relation on sequences:*
Let Q and S be sequences of comparable elements.

$$\text{permut}(Q, S) = \forall e \in \text{concat}(Q, S) : (\text{count}(e, Q) = \text{count}(e, S))$$

Definition 16 *Exploding a sequence (e.g., a list) into a sequence of singleton sequences:*

$$\text{explode\_list}(Q) = \begin{cases} \text{prepend}(\text{prefix}(1, Q), \text{explode\_list}(\text{tail}(Q))) & \text{if size}(Q) > 0 \\ \textit{empty-sequence} & \text{otherwise} \end{cases}$$

Definition 17 *Flattening (i.e. generalized concatenation) of a sequence of sequences Y:*

$$\text{flatten}(Y) = \begin{cases} \text{concat}(y_1, \text{flatten}(\text{tail}(Y))) & \text{if size}(Y) > 0 \\ \textit{empty-sequence} & \text{otherwise} \end{cases}$$

Definition 18 *Splitting a sequence (e.g., a list):*
$\text{split\_list}(Q) = R, S$
such that $\text{size}(R) = \lceil \text{size}(Q)/2 \rceil$, $\text{size}(S) = \lfloor \text{size}(Q)/2 \rfloor$, *and* $Q = \text{concat}(R, S)$.

Definition 19 *Applying a given function to corresponding elements of two sequences (e.g., lists):*

$$\text{map2\_fill}(f, Q, R) = \begin{cases} \text{prepend}(f(q_1, r_1), \text{map2\_fill}(f, \text{tail}(Q), \text{tail}(R))) & \text{if size}(Q) > 0 \land \text{size}(R) > 0 \\ \text{prepend}(q_1, \text{map2\_fill}(f, \text{tail}(Q), R)) & \text{if size}(Q) > 0 \land \text{size}(R) = 0 \\ \text{prepend}(r_1, \text{map2\_fill}(f, Q, \text{tail}(R))) & \text{if size}(Q) = 0 \land \text{size}(R) > 0 \\ \textit{empty-sequence} & \text{otherwise} \end{cases}$$

Definition 20 *Headlist and tail of a sequence Q:*
$\text{headlist\_tail}(Q) = \text{prefix}(1, Q), \text{tail}(Q)$
Note that for an empty-sequence Q^0 (i.e. $\text{size}(Q^0) = 0$*),* $\text{headlist\_tail}(Q^0) = Q^0, Q^0$.

Consider a function f such that:
$f : (D_1 \times \cdots \times D_k \times D'_1 \times \cdots \times D'_m) \mapsto (D'_1 \times \cdots \times D'_m)$
Let $x_1 \in D_1$, ..., $x_k \in D_k$, $y_1 \in D'_1$, ..., $y_m \in D'_m$

Definition 21 *Applying a function f n-times:*

$$\text{iter}(n, f, x_1, ..., x_k, y_1, ..., y_m) = \begin{cases} f(x_1, ..., x_k, \text{iter}(n-1, f, x_1, ..., x_k, y_1, ..., y_m)) & \text{if } n > 0 \\ y_1, ..., y_m & \text{if } n = 0 \end{cases}$$