A Mechanically Verified Incremental Garbage Collector

David M. Russinoff

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive, Austin, TX 78759

A Mechanically Verified Incremental Garbage Collector

David M. Russinoff

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive, Austin, TX 78759

Abstract. As an application of a system designed for concurrent program verification, we describe a formalization and mechanical proof of the correctness of Ben-Ari's incremental garbage collection algorithm. The proof system is based on the Manna-Pnueli model of concurrency and is implemented as an extension of the Boyer-Moore prover. The correctness of the garbage collector is represented by two theorems, stating a) that nothing except garbage is ever collected (safety), and b) that all garbage is eventually collected (liveness). We compare our mechanized treatment with several published proofs of the same results.

Keywords. Boyer-Moore prover, concurrent programming, incremental garbage collection, mechanical program verification.

1 Introduction

As software systems continue to grow in size and complexity, and the cost of programming errors increases, software reliability becomes more important and at the same time more difficult to achieve. The advent of parallel processing introduces a new dimension to this problem, calling for more sophisticated methods for assuring program correctness. As a result, there is broadening recognition of the importance of mathematical methods for verifying program specifications.

There is disagreement, however, on the level of formality that is appropriate for program verification. Most correctness proofs are developed and presented in the traditional high-level style that is typical of mathematical journals. But in comparison with the formulas that mathematicians are accustomed to manipulating, computer programs are large and unwieldy. Consequently, these correctness proofs often contain errors that survive the normal review process.

Mechanical theorem provers offer an alternative to hand-generated proofs, with the potential for significant reduction of errors. However, these provers are generally considered to be difficult to use, requiring tedious attention to detail and expertise that is not common to software engineers. Moreover, they are only applicable, even in principle, to programs that are coded in languages with formal semantic definitions. Unfortunately, this requirement excludes all real programming languages that are in common industrial use today.

As an illustration of the complexity that is inherent in parallel programming, and of the alternative approaches to program verification, we shall consider the problem of incremental garbage collection. This is a notoriously difficult problem, originally posed in 1978 by Dijkstra, Lamport, et al. [DLM78] as an exercise in organizing and verifying the cooperation of concurrent processes. They described their experience as follows:

> Our exercise has not only been very instructive, but at times even humiliating, as we have fallen into nearly every logical trap possible ... It was only too easy to design what looked—sometimes even for weeks and to many people—like a perfectly valid solution, until the effort to prove it correct revealed a (sometimes deep) bug.

Their objective in this exercise was to achieve cooperation between two processes that operate concurrently on a shared data structure, an array consisting of a fixed set of *nodes*. Associated with each node is an array of some uniform length, each entry of which is a pointer either to a node, called a *son*, or to the special symbol `NIL`. (In the case of a LISP system, this uniform length is 2.) One process, called the *mutator*, performs the main computation, while another, the *collector*, is dedicated to storage reclamation. Also associated with each node is a *color* field, which is used by the collector for bookkeeping purposes.

An arbitrary number of nodes, comprising an initial segment of the array, are distinguished as *roots*. (These may be thought of as program variables.) A node is *accessible* if it can be reached by following some succession of pointers from a root, and otherwise is *garbage*. The role of the collector is to identify garbage nodes and append them to a linked list called the *free list*. The mutator, which is continually redirecting pointers from accessible nodes to other accessible nodes (thereby producing garbage), may remove nodes from the free list as they are needed.

There are two criteria for program correctness:

1. *Safety*: No accessible node is ever appended to the free list.

2. *Liveness*: Every garbage node is eventually collected.

An important aspect of the model presented in [DLM78], which serves to simplify the analysis, is that both `NIL` and the head of the free list are considered to be root nodes. Thus, there are only three operations by which the data structure may be altered:

1. Redirect a pointer from one accessible node to another accessible node.

2. Append a garbage node to the free list.

3. Change the color of a node.

In particular, the mutator's operation of removing a node from the free list may be implemented as a sequence of operations of type 1. The other two operations are viewed as overhead and intended to be performed by the collector. However, in order to achieve cooperation, some of this overhead must be assumed by the mutator. One of the design goals for the system is to minimize this mutator overhead. Another is to minimize the amount of space dedicated to garbage collection, i.e., the number of node colors available to the collector. Finally, in order to make full use of concurrency, the exclusion constraints between the two processes should be as weak as possible.

The solution to this problem and its proof of correctness that are presented in [DLM78] are quite complicated, involving three colors. A second solution was found later by Ben-Ari [Ben84]. Ben-Ari's algorithm is similar to the original one, but uses only two colors and admits a somewhat simpler correctness proof. Alternate proofs of the same algorithm were subsequently published by Van de Snepscheut [Van87] and Pixley [Pix88]. All of these proofs follow an informal approach, which Ben-Ari defends as follows:

> So as not to obscure the main ideas, the exposition is limited to the critical facets of the proof. A mechanically verifiable proof would need all sorts of trivial invariants ... and elementary transformations of our invariants (... with appropriate adjustments of the indices).

After describing Ben-Ari's algorithm and correctness argument, we shall examine more closely the feasibility and relative merits of a more formal approach using a mechanical proof system.

Following [DLM78], Ben-Ari imposes no constraints on the number of nodes, the number of roots, or the branching factor (i.e., the number of pointers from each node), although it is understood that each of these parameters is a positive integer. He does assume that with respect to the initial state of the array, "all nodes are linked on the free list and all links not so used are pointing to the root `NIL`". All that is assumed about the free list is that its head is a root which, one must suppose, may or may not be distinct from `NIL`. The details of the appending operation are ignored; there are implicit assumptions pertaining to its properties, but these are unclear.

The color field of a node may assume either of the values *black* and *white*. The collector executes the following simple mark-and-sweep algorithm:

1. Color each root black.

2. Examine each pointer in succession. If the source is black and the target is white, color the target black.

3. Count the black nodes. If the result exceeds the previous count (or if there was no previous count), return to Step 2.

4. Examine each node in succession. If a node is white, append it to the free list; if it is black, color it white. Then return to Step 1.

Clearly, the objective of the marking phase of the algorithm, which consists of Steps 1–3, is to blacken all accessible nodes. In order to avoid interference with this process, the mutator is required to assume the overhead of blackening the new target of a pointer immediately after redirecting it. Thus, the mutator's cycle consists of two steps:

A. Select accessible nodes `R` and `Q` and an index `S` into the array of sons of `R`. Assign `Q` as the $\mathtt{S}^{th}$ son of `R`.

B. Color node `Q` black. Return to Step A.

As usual, concurrency is modeled as arbitrary interleaving of instructions. For this purpose, each step of the mutator is taken to be an atomic instruction, as is each iteration within each step of the collector. In particular, arbitrarily many collector instructions may be executed between the mutator's mutation (Step A) and coloring (Step B) operations. This is the main source of difficulty in establishing the correctness of the program.

Ben-Ari's argument for the safety property is based on a predicate BW, defined as follows on the set of all pointers between nodes: a pointer satisfies BW if its source is an accessible black node and its target is a white node. The crux of the proof lies in showing that this predicate is uniformly false upon exiting the marking phase. This requires two lemmas:

Lemma 1 *During Step 2, as long as the total number of black nodes does not exceed the most recent count, if BW holds for some pointer that has already been visited, then this pointer must have been directed by the most recently executed mutator instruction.*

Lemma 2 *During Step 2, as long as the total number of black nodes does not exceed the most recent count, if BW holds for some pointer that has already been visited, then BW must also hold for some pointer that has not yet been visited.*

Each of these lemmas represents an invariant of the program, which is proved by showing that the stated property holds in the initial state and is preserved by every instruction. Consider Lemma 2, which is the more interesting of the two. The following is a close approximation to Ben-Ari's proof:

Clearly, the statement holds upon entering Step 2, since no pointer has yet been visited. It cannot be negated by any iteration of this step, because if the node being visited satisfies BW, then the blackening of the target violates the bound on the number of black nodes. Similarly, the statement cannot be negated by Step B of the mutator. Finally, assume that it holds true immediately prior to execution of Step A, and consider the state that exists after execution of this instruction. Suppose that in this new state there exists a pointer satisfying BW that has already been visited by the collector. According to Lemma 1, it must be the pointer from `R` to `Q` that was just redirected by the mutator, and no such pointer existed in the preceding state. Since `Q` was accessible and white in this preceding state, there must have been a pointer satisfying BW somewhere on a path from a root to `Q`. But this pointer, which remains unchanged in the current state, must not have been visited by the collector, and hence satisfies the conclusion of Lemma 2.□

At the termination of Step 2, if no node has been blackened since the last count, then according to Lemma 2, BW is false for all pointers, and thus all accessible nodes are black. This must be the case, therefore, at the end of the marking phase. The safety property follows easily.

A "proof" of liveness is also included in [Ben84], but this turns out to be fallacious, as observed in [Van87]. Ben-Ari further argues for the correctness of a variation of his algorithm, in which the order of the mutator instructions is reversed. The authors of [DLM78] recount, as an example of a "logical trap" into which they fell, an analogous claim that they made in connection with their original algorithm, which was shown to be untrue before the proof reached publication. Ben-Ari's claim also turns out to be false—counterexamples are given in [Pix88] and [Van87].

As the astute reader has already observed, Ben-Ari's proof of safety is also fallacious—Lemma 2 is false. In Figure 1, we show a counterexample for the case of 5 nodes, 2 roots, and a branching factor of 1. In this example, the coloring of the roots (nodes 1 and 2) is propagated to nodes 4 and 5 during Step 2, and four black nodes are counted in Step 3. This number is unchanged after an iteration of Step 2, when a mutator instruction is executed. At this point, the pointer from node 1, which has already been visited, is the only pointer that satisfies BW.

Note the error in the above proof that is exposed by this example: the pointer that satisfied BW before Step A was executed no longer satisfies it afterwards; it has not been altered, but its source is no longer accessible. Surprisingly, this error was essentially repeated in [Pix88] and once again survived the review process. Apparently, it has gone undetected from 1984 to the present.

Our summary of the history of this problem is not intended as a negative commentary on the capability of those who have contributed to its solution, all of whom are distinguished scientists. Rather, we present this example as an illustration of the inevitability of human error in the analysis of detailed arguments and as an opportunity to demonstrate the viability of mechanical program verification as an alternative to informal proof.

In [Rus90] and [Rus92], we described an early version of a system designed for the verification of concurrent programs, based on the Manna-Pnueli model of concurrency [MaP81,MaP84] and implemented as an extension of the Boyer-Moore prover [BoM79,BoM88]. Here we shall illustrate the utility of this system in formalizing Ben-Ari's garbage collection algorithm and

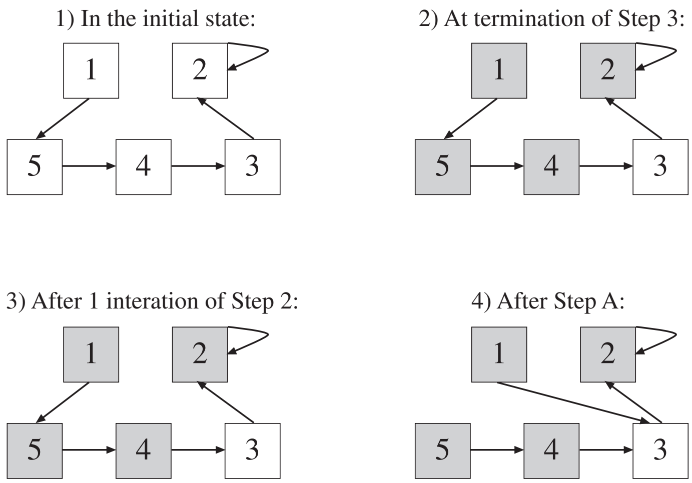

Figure 1: Counterexample to Lemma 2

generating a mechanical proof of its correctness.

After presenting a brief description of the current version of our system in Section 2, we shall develop in Section 3 a formal representation of Ben-Ari's data structure and implement the garbage collector in our concurrent programming language. Its correctness properties are then encoded as formal expressions. In Sections 4 and 5, we describe mechanical proofs of these formulas, adapted from Ben-Ari's informal exposition. Finally, in Section 6, we compare the results of the formal and informal approaches.

2 A Mechanical Concurrent Program Verification System

2.1 The Boyer-Moore Logic

Our verification tool is an extension of the *Nqthm* version of the Boyer-Moore system, which is documented in [BoM88] and includes the extensions described in [BGK89]. This system is founded on a quantifier-free first-order logic with equality and a syntax resembling that of LISP [Ste84]. Thus, terms are constructed from parentheses and symbols denoting variables and functions. These symbols are in turn constructed from a character set that includes numeric and upper-case alphabetic characters. By convention, lower-case alphabetic characters are used to denote metavariables representing functions and terms.

The basic theory contains axioms characterizing four primitive functions:

- `TRUE` and `FALSE` are functions of zero arguments. The constants `(TRUE)` and `(FALSE)`, abbreviated as `T` and `F`, respectively, serve as distinct truth values, as ensured by the axiom $\mathtt{T} \neq \mathtt{F}$.
- `EQUAL` is a binary function. The value of `(EQUAL l r)` is either `T` or `F`, according to whether $\mathtt{l} = \mathtt{r}$.
- `IF` is a ternary function. The value of `(IF t l r)` is the value of `r` if $\mathtt{t} = \mathtt{F}$, and the value of `l` otherwise.

In terms of these primitives, functions are defined corresponding to each of the logical connectives, e.g.,

$$\mathtt{(IMPLIES\ P\ Q)} = \mathtt{(IF\ P\ (IF\ Q\ T\ F)\ T)}.$$

This allows formulas to be encoded as terms, i.e., given any formula ϕ we may construct a term `t` such that

$$\phi \leftrightarrow (\mathtt{t} \neq \mathtt{F})$$

is a theorem. For example, the formula $\mathtt{X} \neq \mathtt{Y} \rightarrow \mathtt{(F\ X\ Y)} = \mathtt{(G\ X)}$ is encoded as the term

$$\mathtt{(IMPLIES\ (NOT\ (EQUAL\ X\ Y))\ (EQUAL\ (F\ X\ Y)\ (G\ X)))}.$$

When a term `t` appears in a context where a formula is expected, it is understood to be an abbreviation for the formula $\mathtt{t} \neq \mathtt{F}$.

Variables occurring in axioms and theorems are understood to be universally quantified. Thus, if a term `t` is a theorem and s is substitution of terms for variables, then the result $\mathtt{t}/s$ of applying s to `t` may be inferred as a theorem by the rule of *instantiation*.

The logic also includes a principle for admitting axioms that define new recursive functions on inductively constructed objects, and a principle of *induction* by which theorems pertaining to these functions may be inferred. Three built-in inductive data types are provided:

- The type *number* formalizes Peano arithmetic through axioms involving the recognizer `NUMBERP`, the constant `(ZERO)` (abbreviated as `0`), and the successor function `ADD1`. Other arithmetic functions are defined in terms of these, including `SUB1` (the inverse of `ADD1`), `LEQ` (the standard partial order), `LESSP` (strict partial order), `ZEROP` (a predicate that fails iff its argument is a non-zero number), `PLUS`, `DIFFERENCE`, `TIMES`, and `QUOTIENT` (the basic binary operations), `DIVIDES` (the standard divisibility relation), and `FIX` (which fixes numbers and coerces non-numbers to `0`).

- The type *cons* formalizes ordered pairs by means of the recognizer `LISTP`, the constructor `CONS`, and the accessors `CAR` and `CDR`. Functions corresponding to other familiar list-processing functions of LISP, such as `MEMBER`, `CADR`, and `LIST`, are defined in terms of these primitives.

- The type *litatom* consists of an object corresponding to each symbol of the logic. The litatom corresponding to the symbol `x` is abbreviated as `'x`.

Variable-free terms that are constructed by means of `CONS` from numbers and litatoms are called *explicit values.* Syntactic conventions similar to those of LISP allow explicit values to be expressed succinctly. For example,

```
(CONS 4 (CONS (CONS 'Z 5) 'NIL))
```

is abbreviated as `'(4 (Z . 5))`. Under these conventions, there is a natural extension of the correspondence between symbols and litatoms that assigns to an arbitrary term `t` an explicit value, denoted `'t`, called its *quotation*, e.g., the quotation of `(PLUS X 3)` is the explicit value denoted by `'(PLUS X 3)`.

Along with this encoding of terms, there is an obvious scheme for encoding variable substitutions as alists (i.e., lists of conses). Thus, the substitution $\{\mathtt{X} \leftarrow \mathtt{2}, \mathtt{Y} \leftarrow \mathtt{3}\}$ is represented by the alist `'((X . 2) (Y . 3))`. This correspondence motivates the definitions of the function `EVAL`, which behaves as a built-in interpreter for the logic, and its companion function `APPLY`. (These are variants of the functions `EVAL$` and `APPLY$`, which are described in [BoM88].) Each of these functions takes two arguments. For a wide class of functions `f`, including most built-in functions and those defined recursively in terms of them,

```
(EQUAL (APPLY 'f (LIST X1 ... Xn)) (f X1 ... Xn))
```

is a theorem, where `n` is the arity of `f` and `X1`, ...`Xn` are distinct variable symbols. We shall call these functions *total*, and any term constructed from them *tame*. (These definitions are somewhat more general than those used in [BoM88] and elsewhere.) If `s` is an alist representing a substitution s that assigns an explicit value to each variable occurring in a tame term `t`, then

$$\texttt{(EVAL 't s)} = \texttt{t}/s$$

is a theorem. For example, in the case `t` = `(PLUS X Y)`, $s = \{\texttt{X} \leftarrow 2, \texttt{Y} \leftarrow 3\}$, we have the theorem

```
(EQUAL (EVAL '(PLUS X Y) '((X . 2) (Y . 3))) (PLUS 2 3)).
```

If the substitution s is not defined on all variables occurring in the term `t`, then `'t` must be replaced in the above theorem by a slightly more complicated expression, which is conveniently represented by means of the LISP *backquote* macro (see [Ste84]). Thus, in the case `t` = `(PLUS X Y)`, $s = \{\texttt{X} \leftarrow 2\}$, the theorem is

```
(EQUAL (EVAL `(PLUS X ',Y) ((X . 2))) (PLUS 2 Y)),
```

where `` `(PLUS X ',Y) `` is an abbreviation for

```
(LIST 'PLUS 'X (LIST 'QUOTE Y)).
```

Another special feature of the logic that is useful for our purpose is the *ordinal* data type, which includes the numbers as a subtype. The ordinals are recognized by the predicate `ORDINALP` and are ordered by the relation `ORD-LESSP`, which extends `LESSP`. Their purpose is to represent well-founded orders that are more complex than the order of the natural numbers, such as lexicographic orders. We shall make use of an ordinal-valued function `LEX`, which behaves as follows: if `l1` and `l2` are lists of numbers of the same length, then

```
(ORD-LESSP (LEX l1) (LEX l2))
```

is true iff `l1` lexicographically precedes `l2`. For convenience in dealing specifically with the lexicographic ordering of pairs, we also define

```
(LEX2-LESSP I1 J1 I2 J2)
            =
(ORD-LESSP (LEX (LIST I1 J1)) (LEX (LIST I2 J2))).
```

2.2 Modeling Concurrent Programs

Our concurrent programming language is based on both the Boyer-Moore logic and the Manna-Pnueli model of concurrency, as presented in [MaP81]. According to this model, a *program* is composed of a finite set of *processes* and is associated with a finite set of *state variables*, consisting of

a) *input variables*, which remain constant throughout execution and are required to satisfy some *input condition*,

b) *global variables*, which are assigned *initial values* expressed in terms of the input variables and may be manipulated by any of the processes, and

c) *program counters*, each of which is associated with some process.

Each process is represented as a graph, consisting of a set of *labels*, which are the admissible values of the corresponding program counter, and one of which is distinguished as the *initial value* of the program counter. These labels are connected by directed *arcs*, each of which is associated with a *precondition* for traversal and a *transition value* corresponding to each global variable of the program.

In our reification of this model, all state variables are symbols of the Boyer-Moore logic and process labels are litatoms. The input condition and all initial values and transition values of global variables are terms, which, for the sake of interpretive semantics, we assume to be tame. We also assume that all variable symbols occurring in these terms are state variables, and moreover, that only input variables occur in the initial values.

As a simple example, we consider a one-process program that computes the factorial function, which is defined by

```
(FACT N)
   =
(IF (ZEROP N) 1 (TIMES N (FACT (SUB1 N)))).
```

This program has one input variable, `N`, with input condition `(NUMBERP N)`, and two global variables, `I` and `R`, with initial values `0` and `1`, respectively. We may think of `I` as a loop variable, which assumes values from `0` to `N`, and `R` as an accumulator, which is initialized to `1` and multiplied by `I` at each

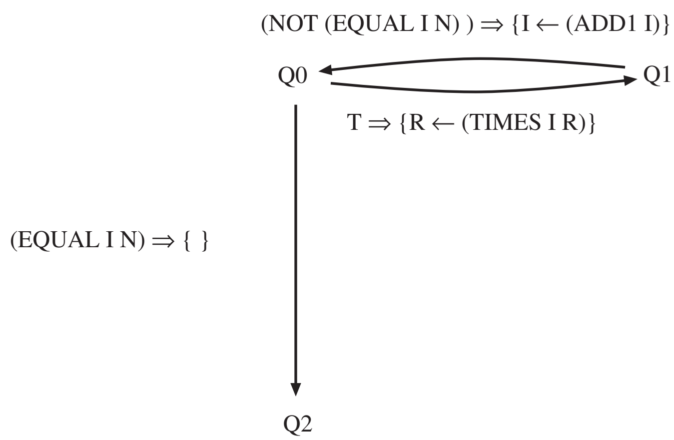

Figure 2: Process `MULT`

iteration. The program's single process has program counter `Q` and three labels, `Q0` (the initial value of `Q`), `Q1`, and `Q2`. These labels are connected by three arcs, as shown in Figure 2. Thus, for example, the arc from `Q1` to `Q0` has the trivial precondition `T` and transition values `I` and `(TIMES I R)` for the globals `I` and `R`.

Formally, processes and programs are encoded as explicit values of the logic. A process is represented as a list of length 2, consisting of the literal atom corresponding to its program counter and a list of objects corresponding to its labels. Each of these objects is itself a list with two members: the label and a list whose members represent the arcs emanating from the label. An arc is encoded as a list of three objects: the terminal label of the arc, the quotation of the arc's precondition, and an alist associating global variables with (the quotations of) their transition values. (A global variable that is its own transition value is omitted from this alist.) According to this scheme, the process of our factorial program is the value of the function `MULT`, defined by

```
(MULT) = '(Q (Q0 ((Q2 (EQUAL I N) ())
                  (Q1 (NOT (EQUAL I N)) ((I . (ADD1 I))))))
             (Q1 ((Q0 T ((R . (TIMES I R))))))
             (Q2 ()))
```

A program is encoded as a list of length 5 whose members correspond to its input variables, global variables, program counters, input condition, and processes. The first member of this list is an alist, associating the input variables with some set of values that satisfy the input condition. (The sole purpose of these values is to allow the system to verify the satisfiability of the input condition.) The next two members of the list are also alists, associating global variables and program counters with their initial values. Thus, our factorial program is represented by

```
'(((N . 5))
  ((I . 0) (R . 1))
  ((Q . Q0))
  (NUMBERP N)
  (,(MULT))).
```

A *state* of a program is a variable substitution that assigns explicit values to its state variables. The *value* of a term `t` with respect to a state s is `t`/s; `t` is *satisfied* by s if `t`/s is true.

An *initial* state of a program is one that satisfies its input condition as well as the term `(EQUAL v i)` for each global variable or program counter `v` with initial value `i`. For example,

```
'((N . 5) (I . 0) (R . 1) (Q . Q0))
```

is an encoding of an initial state of the factorial program. (Here `5` could be replaced by any number.)

An arc associated with some label of a process is *enabled* with respect to a state s if s assigns that label to the process's program counter and s satisfies the precondition of the arc. A *successor* of s is a state derived from s by *traversing* an enabled arc, i.e., by replacing the program counter's value with the arc's terminal label and replacing the value of each global variable with the value with respect to s of its transition value.

An *execution* of a program is an infinite sequence of states, beginning with an initial state, such that each state is followed either by itself or by a successor. The following sequence represents an execution of the factorial program:

```
'((N . 3) (I . 0) (R . 1) (Q . Q0))
'((N . 3) (I . 1) (R . 1) (Q . Q1))
```

```
'((N . 3) (I . 1) (R . 1) (Q . Q0))
'((N . 3) (I . 2) (R . 1) (Q . Q1))
'((N . 3) (I . 2) (R . 2) (Q . Q0))
'((N . 3) (I . 3) (R . 2) (Q . Q1))
'((N . 3) (I . 3) (R . 6) (Q . Q0))
'((N . 3) (I . 3) (R . 6) (Q . Q0))
'((N . 3) (I . 3) (R . 6) (Q . Q2))
'((N . 3) (I . 3) (R . 6) (Q . Q2))
'((N . 3) (I . 3) (R . 6) (Q . Q2))
   .
   .
   .
```

In general, several arcs (either of one process or of different processes) may be enabled with respect to a given state (although this is not illustrated by the example at hand). Moreover, according to our definition of *successor*, an execution may repeat the same state arbitrarily many times. Thus, any number of substantially distinct executions may be possible for the same program and input values. In order to avoid infinite stuttering (repetition of a single state), and to produce a useful model of concurrency, we only consider executions that are *fair* in the following sense: if some process has an enabled arc with respect to all states beyond some point in an execution, then an arc of that process must eventually be traversed at some state beyond that point.

Properties of programs are expressed as formulas constructed by combining terms with the temporal operators "$\Box$" and "$\Diamond$", the semantics of which are defined in [MaP81]. We are only interested in two special classes of properties, which correspond to two types of temporal formulas. A *safety* property (or *invariant*) is represented by an expression of the form

$$\mathtt{r} \rightarrow \Box \mathtt{s} \tag{1}$$

where `r` and `s` are tame terms. Assuming that these terms involve no variables other than the state variables of a program, this formula is satisfied by the program if for every fair execution in which `r` is satisfied by some state, `s` is satisfied by that and every later state. The properties described by these formulas include mutual exclusion, deadlock freedom, and partial correctness. For example, the following represents the partial correctness of our factorial program:

```
□(IMPLIES (EQUAL Q 'Q2)
          (EQUAL K (FACT N))
```

In this example, as in all safety properties that we shall encounter, the antecedent `r` of Formula 1 is the trivial antecedent `T`.

A *Liveness* property (or *eventuality*) IS represented by an instance of the formula

$$\mathtt{r} \rightarrow \Diamond \mathtt{s}. \tag{2}$$

A program satisfies this formula if in every fair execution, any state that satisfies the term `r` is eventually followed by a state satisfying the term `s`. The corresponding properties include termination, accessibility, and responsiveness. For example, the termination property of our example is represented as

```
◇(EQUAL Q 'Q2).
```

We also allow instances of Formulas 1 and 2 that contain variables other than program state variables. Such variables are called *free* and are taken to be universally quantified. More precisely, an expression ϕ that contains free variables is satisfied by a program if for every substitution s that replaces the free variables of ϕ with explicit values, the program satisfies ϕ/s. For example, the following formula, which contains the free variable `X`, states that in every state of every fair execution of the factorial program, `R` is divisible by every positive number less than `I`:

```
□(IMPLIES (AND (NOT (ZEROP X)) (LESSP X I))
          (DIVIDES X R))
```

The procedures by which safety and liveness properties are mechanically verified are described in [Rus90] and [Rus92]. In general, the invariance of a term is proved either a) by showing that it is satisfied in every initial state and is preserved across every arc of every process of the program, or b) as a direct consequence of previously proved invariants. For proving liveness properties, a more complicated method of well-founded measures is used. This involves two terms, which must be supplied by the user: a *measure*, the value of which with respect to each state is an ordinal, and a *helper*, which has a process as its value. To prove an instance of Formula 2, it suffices to show that during any execution,

a) if `r` is satisfied in some state in which `s` is not, then it must continue to be satisfied as long as `s` is unsatisfied, and

b) as long as `r` is satisfied and `s` is not, the value of the measure must never increase, and must decrease whenever an arc of the process that is the value of the helper is traversed.

For the proof of termination of the factorial program, for example, the helper must be `(MULT)`. To determine an appropriate measure, we observe first that at each iteration, the value of `I` increases until it reaches `N`. The primary component of the measure, therefore, is `(DIFFERENCE N I)`. Since the value of this term does not decrease at each step of `(MULT)`, a secondary component is required, which reflects the location of the program counter `Q`. For this purpose, we define a function that computes the location of a value in a list of values:

```
(LOC X L)
    =
(IF (LISTP L)
  (IF (EQUAL X (CAR L)) 0 (ADD1 (LOC X (CDR L))))
    0)
```

The secondary component is `(LOC Q '(Q0 Q1))`, the value of which decreases at each step for which `(DIFFERENCE N I)` remains constant. To combine these two components to form a lexicographic measure, we use the function `LEX`, described in Subsection 2.1:

```
(LEX (LIST (DIFFERENCE N I) (LOC Q '(Q0 Q1)))).
```

2.3 The User Interface

The interface to our system consists of the LISP macros that comprise the interface to the Boyer-Moore prover, along with several others that allow total functions to be constrained, programs to be defined, and program properties to be proved. We shall describe these macros using the format established in [Ste84]. Of the Boyer-Moore macros, we mention only the two that we use most frequently:

- `DEFN` *function args body* `&optional` *hints*

  This command defines a new function symbol according to the principle of recursive definition. Before the definition may be admitted, the prover must establish (using the optional argument *hints*, if provided) the existence of a well-founded measure of the arguments that decreases with respect to each recursive call that occurs in the body.

- `PROVE-LEMMA` *name types term* `&optional` *hints*

  Various heuristics are applied by the prover in an attempt to establish *term* as a theorem. Note that hints may be provided (mainly suggestions of previously proved lemmas to be used in the proof), but no assistance from the user is possible during the course of the proof. If the proof succeeds, then the theorem is entered into the database as a rule of any of several *types*, of which `REWRITE` is the most common.

While there is also a macro that allows the logic to be extended by arbitrary axioms, `DEFN` is preferred, since it is guaranteed not to introduce inconsistency. Another macro with the same guarantee, `CONSTRAIN`, is provided to admit axioms that constrain the behavior of undefined functions. For our purpose, we found it necessary to implement our own variant of `CONSTRAIN`:

- `CONSTRAIN-TOTAL` *name types term pairs* `&optional` *hints*

  A *term* involving one or more new function symbols is presented as an axiom. Before this axiom may be admitted, it must be proved (using *hints*) that some previously existing total functions actually satisfy this constraint. The correspondence between new and old function symbols is given as a list of *pairs*. If the term resulting from *term* under this substitution is proved (using *hints*) and the old functions are shown to be total, then the axiom is admitted, along with axioms representing the totality of the new functions. These axioms are entered as rewrite rules if `REWRITE` is a member of the list of *types* (no other rule type has been implemented).

Processes and programs are coded by means of the following:

- `DEFPROCESS` *name pc* `&rest` *labels*

The symbol *name* is defined as a constant function, the value of which is an encoded process with the program counter *pc* and label encodings *labels*.

- `DEFPROGRAM` *name inputs globals pcs input-cond processes*

  The symbol *name* is defined as a constant function, the value of which is an encoded program with the given components. A syntax checker is invoked to ensure that the definition of *program* is satisfied.

In order to use the theorem prover to verify properties of programs, the temporal formulas representing these properties are encoded as static terms. This involves a set of axioms that characterize fair executions, and the use of `EVAL` to determine whether terms are satisfied by execution states. For example, the two safety properties of the factorial program discussed in Subsection 2.2 are established by proving that the terms

```
(EVAL '(IMPLIES (EQUAL Q 'Q2)
                (EQUAL R (FACT N)))
      s)
```

and

```
(EVAL `(IMPLIES (AND (NOT (ZEROP ',X)) (LESSP ',X I))
                (DIVIDES ',X R))
      s)
```

are true for every state `s` of every fair execution of the program. The procedures for encoding and proving both safety and liveness properties are presented in detail in [Rus90] and [Rus92]; here, we shall describe only the interface to these procedures.

Two macros are provided for proving safety properties, and two others for proving liveness properties. The first three arguments of each of these are the name of an existing program, a proposed name for a property, and a list of rule types, which may include `REWRITE`.

- `PROVE-INVARIANT` *program name types term* `&optional` *hints*

  An attempt is initiated to prove that *term* is an invariant of *program*, by showing that it is satisfied in every initial state and is preserved across every arc. Previously proved invariants may be used in the proof, including those that were specified as rewrite rules and those that are supplied as the optional argument *hints*.

- `PROVE-COROLLARY` *program name types term* `&optional` *hints*

  This is an alternative means of verifying program invariants, which may be used in cases where a term may be proved as a direct consequence of other invariants without examining arcs of the program.

- `PROVE-EVENTUALITY` *program name types*
  antecedent consequent measure helper `&optional` *hints*

  An attempt is initiated to prove a term representing the liveness property

  $$antecedent \rightarrow \Diamond consequent,$$

  using the method of well-founded measures outlined in Subsection 2.2. Along with the optional hints, which must be proved invariants, two other terms are required: an ordinal-valued *measure* of states, i.e., a term satisfying

  $$\texttt{(ORDINALP (EVAL '}measure\texttt{ X))},$$

  and a *helper*, such that

  $$\texttt{(EVAL '}helper\texttt{ X)}$$

  is a process of *program* (for any value of the variable `X`).

- `PROVE-CHAIN` *program name types antecedent consequent chain*

  This macro implements the *chain rule* for liveness properties: if a program satisfies a set of n formulas, $\mathtt{t}_{i-1} \rightarrow \Diamond\mathtt{t}_i$, for $i = 1, \ldots, n$, then it must also satisfy $\mathtt{t}_0 \rightarrow \Diamond\mathtt{t}_n$. In this context, *antecedent* is t_0, *consequent* is t_n, and *chain* is a list of the names of the n intermediate results.

Our formalization of Ben-Ari's algorithm and its properties involves some twenty defined functions, which we introduce by means of `DEFN`, and nine undefined functions, which are specified by three calls to CONSTRAIN-TOTAL. Establishing the required properties of these functions involves over one hundred calls to PROVE-LEMMA. This formalization is described in Section 3.

Once this library of functions and lemmas is in place, the entire proof of program correctness is generated by a total of twenty-seven macro calls, establishing twenty-two program invariants (calls to PROVE-INVARIANT and PROVE-COROLLARY) and five liveness properties (calls to PROVE-EVENTUALITY and PROVE-CHAIN). A complete list of these results is presented in Sections 4 and 5, along with sketches of their mechanical proofs.

3 Formalization of Ben-Ari's Algorithm

3.1 The Data Structure

The array of nodes on which the algorithm operates will be represented as a global variable M of our program. The dimension of the array, the number of roots, and the branching factor will correspond to input variables NODES, ROOTS, and SONS, respectively. Thus, a *node* is an number that is less than NODES, a *root* is a node less than ROOTS, and a *pointer* from a node i is a pair (i j), where j $<$ SONS.

Our implementation of this data structure consists of the formal specification of four functions that access and set the color and son fields of a node. The color of a node is represented by the predicate COLOR: for 0 $\leq$ i $<$ NODES, (COLOR i M) is T or F according to whether node i of M is black or white. The color of a node may be altered by the function SET-COLOR: the value of (SET-COLOR i c M) is the array produced by setting the color of node i of M to c, where c is either T or F.

Similarly, the sons of a node may be accessed and set via the functions SON and SET-SON, respectively. For 0 $\leq$ j $<$ SONS, (SON i j M) returns the jth son of node i, and the result of setting that son to k is the array (SET-SON i j k M).

The functions COLOR, SET-COLOR, SON, and SET-SON are specified by an axiom that has been admitted by means of the CONSTRAIN-TOTAL macro. This axiom is easily shown to be satisfiable by appropriately defined total functions:

Constraint 1

```
(AND (EQUAL (COLOR I M) (COLOR (FIX I) M))
     (EQUAL (SON I J M) (SON (FIX I) (FIX J) M))
     (EQUAL (SON I J (NULL-ARRAY)) 0)
     (EQUAL (COLOR I1 (SET-COLOR I2 C M))
            (IF (EQUAL (FIX I1) (FIX I2))
                C
                (COLOR I1 M)))
     (EQUAL (COLOR I1 (SET-SON I2 J K M))
            (COLOR I1 M))
     (EQUAL (SON I1 J1 (SET-SON I2 J2 K M))
            (IF (AND (EQUAL (FIX I1) (FIX I2))
                     (EQUAL (FIX J1) (FIX J2)))
                K
                (SON I1 J1 M)))
     (EQUAL (SON I1 J (SET-COLOR I2 C M))
            (SON I1 J M)))
```

Thus, all non-numeric indices are coerced to 0. Note that the constraining axiom also guarantees the existence of an array (NULL-ARRAY), in which all pointers are directed to the root 0. We shall make use of this array in Subsection 3.6 in connection with the program's input condition.

Several other total functions are defined in the logic in terms of the constrained functions SON and COLOR. Rather than list their formal definitions here, we describe their behavior informally below. Note that all but the first in this list are predicates, always returning either T or F:

- (BLACKS M K N) returns the number of black nodes i of M in the range $K \leq i < N$. In particular, the total number of black nodes of M is (BLACKS M 0 NODES).

- (CLOSED M NODES SONS) $\Leftrightarrow$ every son of every node of M is a valid node, i.e., for every pointer (i j), where $0 \leq i <$ NODES and $0 \leq j <$ SONS, we have $0 \leq$ (SON i j M $<$ NODES, This property of M will be a useful invariant of our program.

- (ACCESSIBLE i M NODES ROOTS SONS) $\Leftrightarrow$ there is a *path* from some root to node i, i.e., a list of nodes (n_0 n_1 ... n_k), $k \geq 0$, such that $n_0 <$ ROOTS, $n_k = i$, and n_j is a son of n_{j-1} for each $j > 0$.

- (BLACK-ROOTS M r) $\Leftrightarrow$ (COLOR i M) is true for each node i, $0 \leq i \leq$ r. In particular, (BLACK-ROOTS M ROOTS) $\Leftrightarrow$ all roots of M are black.

- (BW i j M) $\Leftrightarrow$ (i j) is a pointer from a black node to a white node, i.e., (COLOR i M) and (NOT (COLOR (SON i j M) M)). Note the difference between this and the stronger predicate BW used in Ben-Ari's proof.

- (EXISTS-BW i1 j1 i2 j2 M SONS) $\Leftrightarrow$ there exists a pointer (i j) from a black node to a white node that lies between (i1 j1) and (i2 j2) with respect to the lexicographic ordering, i.e., satisfying (BW i j M), (NOT (LEX2-LESSP i j i1 j1)), and (LEX2-LESSP i j i2 j2).

- (PROPAGATED M NODES SONS) $\Leftrightarrow$ no pointer satisfies BW, i.e., (NOT (EXISTS-BW 0 0 NODES 0)).

- (BLACKENED M NODES ROOTS SONS k) $\Leftrightarrow$ all accessible nodes among the set $\{$k$, \ldots,$ NODES $- 1\}$ are black. An important lemma, used in the proof of safety (Section 4), states that (BLACK-ROOTS M ROOTS) and (PROPAGATED M NODES SONS) together imply (BLACKENED M NODES ROOTS SONS 0), which says that all accessible nodes are black.

- (PURE i M NODES SONS k) $\Leftrightarrow$ there is no path leading from any black node in the set $\{0, \ldots,$ k $- 1\}$ to node i. This function plays an important role in the proof of liveness (Section 5).

3.2 The Mutator

Recall that at each cycle of the mutator process, three arbitrary selections are made: a source node R, a son index S, and a target node Q, which is to become the Sth son of R. In our formalization, these selections are performed by three undefined functions, COMPUTE-R, COMPUTE-S, and COMPUTE-Q.

Consider first the function COMPUTE-R. Since it is to be constrained to return a number representing a node, one of its arguments should reflect the value of NODES. Since the node that it selects might well depend on the current configuration of the data structure, there should be another argument representing M. But if these were its only two arguments, then COMPUTE-R would be required to return the same value whenever called with the same value of M. That is, the selected node would be completely determined by the current state of M, regardless of the state of any other part of the system. In order to allow other influences on the selection of node R, as well as on the selections of Q and S, we introduce a global variable, P, representing any components of the system that are extraneous to garbage collection (such as the program being executed by the mutator and any data other than M on which it operates). The state of P is passed to COMPUTE-R, along with that of M and the value of NODES.

Similarly, the function COMPUTE-S, which is supposed to return a valid son index, must take, along with M and P, an argument representing the parameter SONS. COMPUTE-Q, which is required to return an accessible node, must be passed all of the arguments of the function ACCESSIBLE:

Constraint 2

```
(IMPLIES (AND (NOT (ZEROP N)) (NOT (ZEROP S)) (NOT (ZEROP R)))
         (AND (NUMBERP (COMPUTE-R M P N))
              (LESSP (COMPUTE-R M P N) N)
              (NUMBERP (COMPUTE-S M P S))
              (LESSP (COMPUTE-S M P S) S)
              (NUMBERP (COMPUTE-Q M P N R S)
              (LESSP (COMPUTE-Q M P N R S) N)
              (ACCESSIBLE (COMPUTE-Q M P N R S) M N R S)))
```

Recall that Ben-Ari's original assumptions also included the accessibility of the source node returned by COMPUTE-R. This assumption turns out to be unnecessary for our proof and is therefore not reflected in Constraint 2.

The mutator's program counter, MU, assumes two values, MU0 and MU1. An arc connects these two nodes in each direction. Along the arc from MU0 to MU1, the three selections are made, the mutation is performed, and the index of the target node is recorded as the value of the global variable Q. In traversing the arc from MU1 to MU0, node Q is colored black.

At any point during execution, the mutator is free to alter the state of P in an unspecified manner, deriving a new value of P based on the current values of M and P. This is formally represented by a loop at each node, replacing the value of P with that of (ALTER-P M P), where ALTER-P is an unconstrained (total) function. Thus, the mutator is encoded as

```
(MUTATOR)
    =
'(MU ((MU0 ((MU0 T ((P . (ALTER-P M P))))
             ;set Sth son of R to Q:
            (MU1 T ((M . (SET-SON (COMPUTE-R M P NODES)
                                  (COMPUTE-S M P SONS)
                                  (COMPUTE-Q M P NODES ROOTS SONS)
                                  M))
                    (Q . (COMPUTE-Q M P NODES ROOTS SONS))))))
      (MU1 ((MU1 T ((P . (ALTER-P M P))))
             ;color node Q:
            (MU0 T ((M . (SET-COLOR Q T M))))))))
```

3.3 The Appending Operation

Before defining the collector process, we must introduce a function that alters the data structure by appending a node to the free list. It would be a simple matter to *define* a function that performs this operation in some reasonable manner. For example, we could arbitrarily decide to take the head of the free list to be the node `(SON 0 0 M)`, and then define, in terms of `SET-SON`, a function that inserts a node `i` at the front of the free list by first directing all of the pointers from node `i` to the current value of `(SON 0 0 M)` and then making `i` the new value of `(SON 0 0 M)`.

For the sake of generality, however, following [DLM78] and [Ben84], we shall avoid defining this operation. That is, our results should be independent of the arbitrary details of the operation, e.g., whether garbage nodes are appended to the front or the end of the free list. On the other hand, it would be useful to *specify* the essential properties of this operation on which the correctness of the program depends, although this has not previously been done (cf. [Ben84,DLM78,Pix88,Van87]). Indeed, such a specification is a prerequisite for a formal proof.

We shall introduce a function `APPEND-TO-FREE` by means of a constraining axiom that embodies four essential properties. Two of these are trivial (although their necessity only became obvious to the author during the course of the mechanical proof):

1. The appending operation never changes the color of a node.

2. The appending operation never transforms a closed array into an unclosed array.

The third property pertains to accessibility. Clearly, a garbage node must become accessible upon being appended to the free list. Furthermore, no node that is already accessible should be rendered garbage by this operation. (While the latter condition has intuitive appeal, its necessity will not become clear until we see the proof of Invariant 21 in Section 5.) It is also desirable, although perhaps less obviously, that no garbage node other than the one being appended may become accessible as a side effect. It is the safety property of the program that depends on this last assumption. To see this, suppose that in the act of appending a garbage node `i`, another garbage node `j` were to become accessible. If `j` were examined at some later point during the same appending phase and found to be white, then it would be appended, even though already accessible (assuming the mutator had not altered the state in the meantime).

As we shall see, these assumptions concerning accessibility will only be required in contexts in which the node being appended is known to be garbage. This is fortunate, since no reasonably efficient implementation could be expected to conform to them if applied to a node that is already accessible. (Consider, for example, the candidate operation of inserting a node at the head of the free list, as described above.) Thus, they may be summarized as follows:

3. In appending a garbage node, only that node becomes accessible, and no accessible node becomes garbage.

The final property of the appending operation pertains to garbage nodes that remain garbage:

4. In appending a garbage node, no pointer from any other garbage node is altered.

It is somewhat surprising that we need any such restriction concerning pointers between garbage nodes, but the proof of liveness depends on it. It will be necessary to prove that any node `i` that is garbage at the beginning of an appending phase and that is not collected (but rather is whitened) during that appending phase will be collected during the next appending phase. But suppose that as a result of an appending operation that occurs during the first appending phase, `i` were to become a son (or more generally, a descendant) of some black garbage node `j` that remains black upon entering the subsequent marking phase. This might occur in the absence of our fourth assumption if, for example, `j` were initially accessible and black, but then a) whitened by the collector, b) blackened by the mutator, c) rendered garbage by the mutator, and finally d) made a parent of `i` by the collector. In this case, as long as the pointer from `j` to `i` were not altered by the mutator in the meantime, `i` would be blackened during the marking phase and consequently not collected during the next appending phase.

These four properties are formally expressed by the following axiom:

Constraint 3

```
(IMPLIES
  (AND (NOT (ZEROP N)) (NOT (ZEROP S))
       (NOT (ZEROP R)) (LEQ R N)
       (NUMBERP I) (LESSP I N))
  (AND (EQUAL (COLOR J (APPEND-TO-FREE I M N R S))
              (COLOR J M))
       (IMPLIES (CLOSED M N S)
                (CLOSED (APPEND-TO-FREE I M N R S) N S))
       (IMPLIES
         (NOT (ACCESSIBLE I M N R S))
         (IFF (ACCESSIBLE J (APPEND-TO-FREE I M N R S) N R S)
              (OR (EQUAL I J) (ACCESSIBLE J M N R S))))
       (IMPLIES
         (AND (NOT (ACCESSIBLE I M N R S))
              (NUMBERP J) (NOT (EQUAL J I))
              (NOT (ACCESSIBLE J M N R S)))
         (EQUAL (SON J K (APPEND-TO-FREE I M N R S))
                (SON J K M)))))
```

3.4 The Collector

In addition to the global variable `M` and the input variables `NODES`, `ROOTS`, and `SONS`, the collector process accesses seven other global variables: `BC` and

OBC, used for counting black nodes, and the loop variables I, J, K, L, and H. Its formal definition is

```
(COLLECTOR)
     =
'(CHI ( ;blacken roots:
       (CHI0 ((CHI1 (EQUAL K ROOTS) ((I . 0)))
              (CHI0 (NOT (EQUAL K ROOTS))
                ((M . (SET-COLOR K T M)) (K . (ADD1 K))))))

        ;propagate coloring:
       (CHI1 ((CHI4 (EQUAL I NODES) ((BC . 0) (H . 0)))
              (CHI2 (NOT (EQUAL I NODES)) NIL)))
       (CHI2 ((CHI1 (NOT (COLOR I M)) ((I . (ADD1 I))))
              (CHI3 (COLOR I M) ((J . 0)))))
       (CHI3 ((CHI1 (EQUAL J SONS) ((I . (ADD1 I))))
              (CHI3 (NOT (EQUAL J SONS))
                ((M . (SET-COLOR (SON I J M) T M))
                 (J . (ADD1 J))))))

        ;count black nodes and compare with previous count:
       (CHI4 ((CHI6 (EQUAL H NODES) NIL)
              (CHI5 (NOT (EQUAL H NODES)) NIL)))
       (CHI5 ((CHI4 (NOT (COLOR H M)) ((H . (ADD1 H))))
              (CHI4 (COLOR H M) ((BC . (ADD1 BC)) (H . (ADD1 H))))))
       (CHI6 ((CHI1 (NOT (EQUAL OBC BC)) ((OBC . BC) (I . 0)))
              (CHI7 (EQUAL OBC BC) ((L . 0)))))

        ;append white nodes to free list:
       (CHI7 ((CHI0 (EQUAL L NODES) ((BC . 0) (OBC . 0) (K . 0)))
              (CHI8 (NOT (EQUAL L NODES)) ())))
       (CHI8 ((CHI7 (COLOR L M)
               ((M . (SET-COLOR L F M)) (L . (ADD1 L))))
             (CHI7 (NOT (COLOR L M))
                ((M . (APPEND-TO-FREE L M NODES ROOTS SONS))
                 (L . (ADD1 L))))))))
```

Thus, the program counter CHI assumes nine values. The first seven of these comprise the *marking phase*, which is further partitioned into the *root-blackening*, *propagation*, and *counting phases* as indicated by embedded comments. The remaining two labels comprise the *appending phase*.

At CHI0, node K is blackened, where K ranges from its initial value 0 through ROOTS−1. When the test (EQUAL K ROOTS) succeeds, the arc from CHI0 to CHI1 is traversed and the loop variable I is initialized to 0 for the propagation phase.

For each value of I, $0 \leq$ I $<$ NODES, the color of node I is examined at CHI2. If it is black, i.e., if (COLOR I M) is true, then each of its sons, (SON I J M), J $=$ 1, …, SONS, is colored at CHI3. When the test (EQUAL I NODES) succeeds at CHI1, the variables BC and H are set to 0 and control is passed to CHI4.

The purpose of CHI4, CHI5, and CHI6 is to count the black nodes. At CHI5, as H is incremented, the value of BC is incremented for each value of H for which (COLOR H M) is true. When the test (EQUAL H NODES) succeeds (i.e., all nodes have been examined), the count stored in BC is compared with the previous count, which is stored in OBC (old black count). The test (EQUAL OBC BC) at CHI6 determines whether to repeat the propagation phase (after storing the new count in OBC and reinitializing I) or to proceed to the appending phase (by initializing the loop variable L and passing control to CHI7).

Upon traversing the arc from CHI6 to CHI7, all accessible nodes are assumed to be black. For L $=$ 0, …, NODES $-$ 1, the color of node L is examined at CHI8. If it is white ((COLOR L M) $=$ F), then it is taken to be garbage and appended to the free list; otherwise, it is simply colored white. When all nodes have been examined, control is returned to CHI0.

3.5 The Program

The formal definition of the garbage collector program is

```
(GC)
 =
'( ;input variables:
  ((NODES . 5) (ROOTS . 3) (SONS . 2)
   (M0 . (NULL-ARRAY)) (P0 . 0))
```

```
 ;global variables:
((M . M0) (P . P0) (Q . 0) (BC . 0) (OBC . 0)
 (I . 0) (J . 0) (K . 0) (L . 0) (H . 0))
 ;program counters:
((MU . MU0) (CHI . CHI0))
 ;input condition:
(AND (NOT (ZEROP SONS)) (NOT (ZEROP ROOTS))
     (NOT (ZEROP NODES)) (LEQ ROOTS NODES)
     (CLOSED M0 NODES SONS))
 ;processes:
(,(MUTATOR) ,(COLLECTOR)))
```

Each of the state variables of Program `GC` has already been discussed, except for the input variables `M0` and `P0`. Their purpose is to provide initial values for the global variables `M` and `P`. Note that `M0`, the initial state of `M`, is constrained by the input condition to be a closed array, while the initial value `P0` of `P` is completely arbitrary.

Since the program counter `CHI` is initialized to `CHI0`, it is important that the loop variable `K` has the proper initial value, `0`. The initial value `0` of the counting variable `OBC` is also significant. All remaining global variables are initialized arbitrarily to `0`.

Recall that the admissibility of a program definition requires that the sample values supplied for the input variables satisfy the input condition. This was the motivation for the introduction of the function `NULL-ARRAY` in Constraint 1—it is a consequence of this axiom that `(NULL-ARRAY)` is a closed array.

4 Proof of Safety

In this section, we derive the main safety property of program `GC`: *no accessible node is ever appended to the free list.* According to the definition of `(COLLECTOR)`, the appending operation is performed only when the program counter `CHI` has the value `CHI8`. It is applied to node `L`, but only in the event that `(COLOR L M)` = `F`. Thus, the desired result may be stated formally as the invariant

```
□(IMPLIES (AND (EQUAL CHI 'CHI8)
```

```
            (ACCESSIBLE L M NODES ROOTS SONS))
        (COLOR L M).
```

As seen below, the proof involves a total of 20 program invariants.

4.1 Variable Bounds and Closure

The following five invariants give obvious bounds on the loop variables `I`, `J`, `K`, `H`, and `L` of the collector process:

Invariant 1

```
□(AND (NUMBERP I)
      (NOT (LESSP NODES I))
      (IMPLIES (MEMBER CHI '(CHI2 CHI3)) (LESSP I NODES)))
```

Invariant 2

```
□(AND (NUMBERP J)
      (NOT (LESSP SONS J))
      (IMPLIES (NOT (EQUAL J SONS)) (LESSP J SONS)))
```

Invariant 3

```
□(AND (NUMBERP K)
      (NOT (LESSP ROOTS K))
      (IMPLIES (NOT (EQUAL K ROOTS)) (LESSP K ROOTS)))
```

Invariant 4

```
□(AND (NUMBERP H)
      (NOT (LESSP NODES H))
      (IMPLIES (EQUAL CHI 'CHI5) (LESSP H NODES))
      (IMPLIES (EQUAL CHI 'CHI6) (EQUAL H NODES)))
```

Invariant 5

```
□(AND (NUMBERP L)
      (NOT (LESSP NODES L))
      (IMPLIES (EQUAL CHI 'CHI8) (LESSP L NODES)))
```

A similar bound on the index `Q` is easily derived from Constraint 1:

Invariant 6

```
□(AND (NUMBERP Q) (LESSP Q NODES))
```

The invariance of the closure property of the array `M`, `(CLOSED M NODES SONS)`, may be derived as a consequence of Invariants 5 and 6. To see this, first recall that the input condition of `GC` guarantees the initial value `M0` of `M` is closed. It may be shown, using Constraint 1 and Invariant 6, that closure is preserved by the `SET-SON` operation performed by the mutator, as well as by any transition that merely colors a node. The only remaining arc that alters `M` is the one involving `APPEND-TO-FREE`. But by Constraint 3 and Invariant 5, this arc preserves closure as well. Thus, we have

Invariant 7

```
□(CLOSED M NODES SONS)
```

4.2 Counting Black Nodes

The variable `OBC` is initially set to `0` and later represents the most recent count of black nodes. We would like to show that at any point during the propagation phase, the value of `OBC` does not exceed the total number of black nodes, `(BLACKS M 0 NODES)`.

To this end, we prove first that during the counting phase, the value of `BC`, the current count, does not exceed `(BLACKS M 0 H)`, the number of black nodes that have already been examined. This inequality trivially holds upon entering the counting phase (when `BC` is set to `0`), and clearly cannot be negated by the mutator, which never alters `BC` and can only increase `(BLACKS M 0 H)`. It must also be shown that the equality is preserved by the collector across each of the two arcs from `CHI5` to `CHI4`. These arcs correspond to the two cases `(NOT (COLOR H M))` and `(COLOR H M)`. In the first case, neither `BC` nor `(BLACKS M 0 H)` is altered; in the second case, since `(BLACKS M 0 (ADD1 H))` = `(ADD1 (BLACKS M 0 H))`, the two values are simultaneously incremented. Thus, we have

Invariant 8

```
□(IMPLIES (MEMBER CHI '(CHI4 CHI5))
          (NOT (LESSP (BLACKS M 0 H) BC)))
```

It follows that when the counting is completed, `BC` does not exceed the total number of blacks:

Invariant 9
```
□(IMPLIES (EQUAL CHI 'CHI6)
          (NOT (LESSP (BLACKS M 0 NODES) BC)))
```

OBC is always 0 at CHI0 and is set to the value of BC in traversing the arc from CHI6 to CHI1. Thus, the desired bound on OBC holds upon entering the propagation phase. Since it is easily shown to be preserved by the collector within the propagation phase as well as by the mutator, we have

Invariant 10
```
□(IMPLIES (MEMBER CHI '(CHI0 CHI1 CHI2 CHI3))
          (NOT (LESSP (BLACKS M 0 NODES) OBC))
```

During the counting phase, another upper bound on OBC is in effect: the sum of BC (the number of black nodes that have already been counted) and (BLACKS M H NODES) (the number of black nodes that have yet to be counted). Since BC and H are initialized to 0 in traversing the arc from CHI1 to CHI4, it follows from Invariant 10 that this inequality holds upon entering the counting phase. Its invariance is then easily established:

Invariant 11
```
□(IMPLIES (MEMBER CHI '(CHI4 CHI5 CHI6))
          (LEQ OBC (PLUS BC (BLACKS M H NODES))))
```

Two other inequalities concerning BC and OBC will be needed in Section 5. The first of these follows from Invariants 8 and 9, and the second is an immediate consequence of Invariant 11:

Invariant 12
```
□(NOT (LESSP NODES BC))
```

Invariant 13
```
□(IMPLIES (EQUAL CHI 'CHI6) (NOT (LESSP BC OBC))).
```

4.3 Coloring Accessible Nodes

The objective of the collector's marking phase is to blacken all accessible nodes. As observed in Subsection 3.1, the formal statement of this goal is `(BLACKENED M NODES ROOTS SONS 0)`. Once we have shown that this term is satisfied upon entering the appending phase, the safety result will follow easily.

The first step of the marking process, the root-blackening phase, is described in terms of the function `BLACK-ROOTS`. The following result is easily proved:

Invariant 14

```
□(IMPLIES (MEMBER CHI '(CHI0 CHI1 CHI2 CHI3 CHI4 CHI5 CHI6))
          (BLACK-ROOTS M (IF (EQUAL CHI 'CHI0) K ROOTS)))
```

In particular, `(BLACK-ROOTS M ROOTS)` holds, i.e., all roots are black, upon traversing the arc from `CHI0` to `CHI1`, and this remains true throughout the marking phase.

In order to establish the success of the marking phase, it would now suffice to show that upon termination of this phase, no accessible black node has a white son. To attempt to prove this directly is the approach taken in [Ben84], where we have seen it fail. The point of our departure from this approach is our weakening of the predicate `BW`. Thus, following [Van87], we prove a stronger result: upon termination of the marking phase, no black node, *accessible or not*, has a pointer to a white son.

Our goal, then, is to show that `(PROPAGATED M NODES SONS)` is true at the end of the marking phase. To this end, we shall derive variants of Ben-Ari's Lemmas 1 and 2 involving our predicate `BW`.

Note that the order in which the son pointers are examined by our collector process during the propagation phase is the lexicographic order defined by the predicate `LEX2-LESSP`. At node `CHI3`, the pointer `(I J)` is visited. At `CHI1` or `CHI2`, the pointer that will next be visited is `(I 0)`. Hence, at any point during the propagation phase, a given pointer `(y z)` has already been visited iff

```
(LEX2-LESSP y z I (IF (EQUAL CHI 'CHI3) J 0))
```

is satisfied.

With this in mind, we may restate Ben-Ari's Lemma 1 as follows:

Invariant 15

```
□(IMPLIES (AND (MEMBER CHI '(CHI1 CHI2 CHI3))
               (EQUAL (BLACKS M 0 NODES) OBC)
               (LESSP Z SONS)
               (LEX2-LESSP Y Z I (IF (EQUAL CHI 'CHI3) J 0))
               (BW Y Z M))
          (AND (EQUAL MU 'MU1) (EQUAL (SON Y Z M) Q)))
```

Note that the variables Y and Z occurring in this term are free, and therefore, in effect, universally quantified. Thus, according to Invariant 15, as long as the collector is in the propagation phase and the number of black nodes is the value of OBC, if there exists a pointer (Y Z) satisfying BW that has already been visited, then the mutator must be at MU1 and the white node must be node Q.

The proof of this invariant depends on Invariant 10, which guarantees that the stated property cannot be negated by any coloring operation. It is thus easily shown to be preserved across all arcs of the collector as well as the mutator.

The following more useful invariant, which involves no free variables, is a direct corollary of Invariant 15:

Invariant 16

```
□(IMPLIES (AND (MEMBER CHI '(CHI1 CHI2 CHI3))
               (EQUAL (BLACKS M 0 NODES) OBC)
               (EXISTS-BW 0 0 I (IF (EQUAL CHI 'CHI3) J 0) M SONS))
          (EQUAL MU 'MU1))
```

Our version of Ben-Ari's Lemma 2 states that during the propagation phase, as long as the number of black nodes is the value of OBC, if there is a pointer from a black node to a white node that has already been visited, then there must also be a pointer from a black node to a white node that has not yet been visited:

Invariant 17

```
□(IMPLIES
   (AND (MEMBER CHI '(CHI1 CHI2 CHI3))
        (EQUAL (BLACKS M 0 NODES) OBC)
        (EXISTS-BW 0 0 I (IF (EQUAL CHI 'CHI3) J 0) M SONS))
   (EXISTS-BW I (IF (EQUAL CHI 'CHI3) J 0) NODES 0 M SONS))
```

The proof of Invariant 17 follows the outline of Ben-Ari's proof, as presented in Section 1. The most interesting case is the arc of the mutator from `MU0` to `MU1`. Suppose that Invariant 17 is satisfied by a state s_0 in which `MU` = `MU0` and let s_1 be a state produced by traversing the arc to `MU1`. Assume the the hypotheses of Invariant 17 are satisfied by s_1. It must be shown that the conclusion is also satisfied.

First note that the only pointer for which the value of our `BW` predicate changes in traversing this arc is the one that is altered by the `SET-SON` operation. (This is not true for Ben-Ari's version of this predicate.) By Invariant 16, the third hypothesis of Invariant 17 must not be satisfied by s_0. Thus, there is some pointer that has been examined by the collector and satisfies `BW` with respect to s_1 but not s_0. This pointer must be the one that is being redirected, and its new target must be white. But according to Constraint 2, this target node is accessible in state s_0. It follows from Invariant 14 that some pointer satisfies `BW` in state s_0. This pointer must already have been examined by the collector. Since it is not the one being redirected, it must also satisfy `BW` with respect to s_1. Hence, the conclusion of Invariant 17 is true in s_1, and the proof is complete.

Suppose now that `(EQUAL (BLACKS M 0 NODES) OBC)` is true at the end of the propagation phase, i.e., as the arc from `CHI1` to `CHI4` is traversed. Since `I` = `NODES`, it follows that the conclusion of Invariant 17 is false in this state. But then the third hypothesis must also be unsatisfied, hence `(PROPAGATED M NODES SONS)` holds, and we may conclude `(BLACKENED M NODES ROOTS SONS 0)`. We would like to establish the following invariant:

Invariant 18

```
□(IMPLIES (AND (MEMBER CHI '(CHI4 CHI5 CHI6))
               (EQUAL OBC (PLUS BC (BLACKS M H NODES))))
          (BLACKENED M NODES ROOTS SONS 0))
```

We have already shown by the above remarks that Invariant 18 holds upon entering the counting phase, since `BC` = `H` = `0`. Since it is trivially preserved across both arcs emanating from `CHI4`, the only arcs of the collector that we need consider are those from `CHI5` to `CHI4`. There are two cases, according to the value of `(COLOR H M)`: if node `H` is black, then `(BLACKS M H NODES)` is decremented, while `BC` is incremented; if `H` is white, then `(BLACKS M H NODES)` is unchanged, as is `BC`.

It remains to be shown that Invariant 18 is preserved by the mutator. First note that since the mutator does not alter the value of `OBC` and can only increase that of `(PLUS BC (BLACKS M H NODES))`, it follows from Invariant 11 that the hypothesis `(EQUAL OBC (PLUS BC (BLACKS M H NODES)))` cannot become true as a result of a mutator operation. Next, observe that since the accessibility of the node returned by `COMPUTE-Q` is guaranteed by Constraint 2, no inaccessible node may become accessible by the mutator's `SET-SON` operation. Finally, since the mutator never whitens a node, it always preserves the conclusion of Invariant 18.

4.4 The Appending Phase

Upon entering the appending phase (via the arc from `CHI6` to `CHI7`), we have `OBC` = `BC` and (by Invariant 4) `H` = `NODES`. Thus, `(BLACKS M H NODES)` = `0` and the hypothesis of Invariant 18 is satisfied. Since `L` = `0`, the conclusion may be expressed as `(BLACKENED M NODES ROOTS SONS L)`. We would like to show that this remains true throughout the appending phase:

Invariant 19

```
□(IMPLIES (MEMBER CHI '(CHI7 CHI8))
          (BLACKENED M NODES ROOTS SONS L))
```

As in the proof of Invariant 18, we need only consider the action of the collector, and hence we may restrict our attention to the two arcs from `CHI8` to `CHI7`. By Constraint 3, neither of these arcs may affect either the accessibility or the color of any node other than node `L`. Since both arcs replace `L` with `(ADD1 L)`, neither of them can negate `(BLACKENED M NODES ROOTS SONS L)`, and Invariant 19 follows.

Finally, the safety property of `GC` is a direct consequence of Invariant 19 and the definition of `BLACKENED`:

Invariant 20

```
□(IMPLIES (AND (EQUAL CHI 'CHI8)
               (ACCESSIBLE L M NODES ROOTS SONS))
          (COLOR L M))
```

5 Proof of Liveness

In this section, we describe the proof of the liveness property of `GC`: *every node of* `M` *eventually becomes accessible.* This goal is expressed formally by

```
    (AND (NUMBERP Z) (LESSP Z NODES))
```
$\rightarrow \Diamond$`(ACCESSIBLE Z M NODES ROOTS SONS)`

(in which `Z` occurs as a free variable). The proof, which rests on the results of Section 4, requires two additional invariants and four intermediate eventualities.

The macro `PROVE-EVENTUALITY`, as described in Section 2, is used to derive these four eventualities. In each call to this macro, the *helper* argument is simply the `COLLECTOR` process, while the values of the *measure* argument are expressions based on lexicigraphic orderings of natural numbers, defined in terms of the function `LEX`, which is described in Section 2.

Finally, our ultimate goal, as stated above, will be derived by applying the macro `PROVE-CHAIN` to these other eventualities. Thus, these other four formulas will have the form $\mathtt{t}_{i-1} \rightarrow \Diamond\mathtt{t}_i$, for $i = 1, \ldots, 4$, where $\mathtt{t}_0$ is the antecedent `(AND (NUMBERP Z) (LESSP Z NODES))` and $\mathtt{t}_4$ is the conclusion `(ACCESSIBLE Z M NODES ROOTS SONS)`.

5.1 Accessibility of `Q`

We shall require the following invariant, which appears at first glance to be trivial, but is seen upon closer examination to depend on nearly all that we have done up to this point:

Invariant 21
$\Box$`(ACCESSIBLE Q M NODES ROOTS SONS)`

Since the input condition of `GC` implies that `0` is a root and therefore accessible, this term is clearly satisfied in the initial state, in which `Q` = `0`. It is preserved by the mutator arc from `MU0` to `MU1`, which is the only arc that alters the value of `Q`. This follows from Constraint 2, which ensures that the node returned by `COMPUTE-Q` is accessible, and the observation that an accessible node cannot be rendered inaccessible by redirecting a pointer to it.

The accessibility of node Q is also preserved by the other mutator arc, as well as by all coloring operations performed by the collector. The only remaining operation of the collector that affects any of the variables occurring in Invariant 21 is the appending operation at CHI8. To complete the proof, we must invoke Constraint 3, which guarantees, *since* L *is accessible*, that no garbage is created by this operation. Note that this argument reveals an interesting dependency: the invariance of the accessibility of Q depends on the safety property of the program. It also finally demonstrates the necessity of the full form of Property 3 of the appending operation, as stated in Subsection 3.4.

5.2 Collection of Pure Nodes

If a garbage node is white at the beginning of an appending phase, then it will be collected during that phase. If it is black at the beginning of an appending phase, then it will be whitened during that phase and will remain white until the subsequent appending phase, during which it will be collected.

It is not the case, however, that whiteness of a node that is white at the end of an appending phase is sufficient to guarantee that it is collected during the next appending phase. Recall that (PURE i M NODES SONS NODES) means that there is no path leading from a black node to node i. In this case, we shall say that node i is *pure*. As we shall prove in the next subsection, a node that is black at the beginning of an appending phase is not only white, but pure, at the end of that phase. It is this property that ensures that it will remain white until the next appending phase.

In the present subsection, we shall show that a garbage node that is pure at any point during a marking phase will eventually be collected. This claim is decomposed into two eventualities. The first of these states that if, during an appending phase, Z is a white node that has not yet been examined (i.e., L $\leq$ Z), then either Z is already accessible or Z will eventually be collected (in fact, this will occur during the current appending phase):

Eventuality 1

```
(OR (ACCESSIBLE Z M NODES ROOTS SONS)
    (AND (MEMBER CHI '(CHI7 CHI8))
         (NOT (COLOR Z M))
         (NUMBERP Z) (LEQ L Z) (LESSP Z NODES)))
```

$\rightarrow \Diamond$`(ACCESSIBLE Z M NODES ROOTS SONS)`

As outlined in Subsection 2.2, the proof of this formula involves a) showing that if the antecedent is satisfied in some execution state, then it continues to be satisfied until a state is reached in which the conclusion is satisfied, and b) constructing an ordinal-valued measure `m` such that as long as the antecedent is satisfied and the conclusion is not, the value of `m` never increases and strictly decreases with every action of the collector.

To prove a), suppose that in some state, `Z` is inaccessible and

```
(AND (MEMBER CHI '(CHI7 CHI8))
     (NOT (COLOR Z M))
     (NUMBERP Z) (LEQ L Z) (LESSP Z NODES)))
```

is true. By Invariant 21, `Z` $\neq$ `Q` and hence the above term cannot be rendered false by the mutator. The only way that the collector may negate this term is by crossing an arc that increments `L`, starting from a state in which `L` = `Z`. But this must then be the arc with precondition `(NOT (COLOR L M))`, which has the effect of appending `Z` to the free list, thus (by Constraint 3) causing `Z` to become accessible.

Next, we construct the measure `m`. The primary component of `m` is the difference between `NODES` and `L`, which decreases with each execution of the loop of the appending phase. Within each loop, for constant `L`, the program counter proceeds from `CHI7` to `CHI8`. Hence, `m` is the term

```
(LEX (LIST (DIFFERENCE NODES L) (LOC CHI '(CHI8 CHI7)))).
```

It is trivial to complete the proof of Eventuality 1 by verifying that this measure satifies the general requirement stated above.

The second formula that must be proved to establish our claim concerning the collection of pure nodes states that a garbage node that is pure at any point during a marking phase must eventually satisfy the hypothesis of Eventuality 1:

Eventuality 2

```
(OR (ACCESSIBLE Z M NODES ROOTS SONS)
    (AND (NUMBERP Z) (LESSP Z NODES)
         (MEMBER CHI '(CHI0 CHI1 CHI2 CHI3 CHI4 CHI5 CHI6))
```

```
            (PURE Z M NODES SONS NODES)))
→ ◇(OR (ACCESSIBLE Z M NODES ROOTS SONS)
       (AND (MEMBER CHI '(CHI7 CHI8))
            (NOT (COLOR Z M))
            (NUMBERP Z) (LEQ L Z) (LESSP Z NODES)))
```

To prove this formula, suppose that with respect to some execution state,

```
(AND (NUMBERP Z) (LESSP Z NODES)
     (MEMBER CHI '(CHI0 CHI1 CHI2 CHI3 CHI4 CHI5 CHI6))
     (PURE Z M NODES SONS NODES))
```

is satisfied, where `Z` is some garbage node. One arc that fails to preserve the truth of this term is the arc from `CHI6` to `CHI7`. But since every pure node is white, traversing this arc to `CHI7` renders the conclusion of Eventuality 2 true.

All other arcs of the program preserve the truth of the above term. To see this, we need only consider arcs that may negate the term `(PURE Z M NODES SONS NODES)` by altering the value of `M`. The only such arc in the marking phase of the collector is the loop at `CHI3`, which colors the node `(SON I J)`. To show that `Z` cannot become impure as a result of this operation, we observe that a pure node cannot become impure by the blackening of a son of a black node, and invoke the following invariant, which follows easily from Constraint 1:

Invariant 22

```
□(IMPLIES (EQUAL CHI 'CHI3) (COLOR I M))
```

We must also consider the two arcs of the mutator that affect `M`. Since `Q` is accessible (by Invariant 21) and `Z` is not, `Z` cannot be rendered impure by the coloring operation of the arc from `MU1` to `MU0`. Similarly, we may dispose of the arc from `MU0` to `MU1` by observing that no path can be created from a black node to the inaccessible `Z` be redirecting a pointer to the accessible node `Q`.

A rather complicated measure is needed to establish Eventuality 2:

```
(LEX (LIST (DIFFERENCE NODES OBC)
           (PHASE CHI '((CHI6 CHI5 CHI4)
```

```
                    (CHI3 CHI2 CHI1)
                    (CHI0)))
       (DIFFERENCE NODES H)
       (LOC CHI '(CHI6 CHI5 CHI4))
       (DIFFERENCE NODES I)
       (LOC CHI '(CHI3 CHI2 CHI1))
       (DIFFERENCE SONS J)
       (DIFFERENCE ROOTS K)))
```

The primary component of this measure, `(DIFFERENCE NODES OBC)`, is based on the observation that the value of `OBC`, which is bounded by `NODES`, decreases on successive loops through the marking phase. The secondary component involves a new function `PHASE`, defined by

```
(PHASE X L)
    =
(IF (LISTP L)
  (IF (MEMBER X (CAR L)) 0 (ADD1 (PHASE X (CDR L))))
    0)
```

Thus, the value of

```
(PHASE CHI '((CHI6 CHI5 CHI4) (CHI3 CHI2 CHI1) (CHI0)))
```

decreases as the program counter `CHI` moves first from the root-blackening phase into the propagation phase, and then into the counting phase. Each of the remaining components of our measure represents a quantity that decreases during one of these three phases: the third and fourth correspond to the counting phase, the next three to the propagation phase, and the last to the root-blackening phase. It is easily verified that throughout the marking phase, the value of this measure is unaffected by the mutator and is decreased by every transition of the collector. This completes the proof of Eventuality 2.

5.3 Purification of Garbage

To complete the proof of the liveness property, we show that any garbage node eventually either is pure during a marking phase (in which case we have shown that it is collected) or is collected. This claim is also established by proving two eventualities. The first states that the claim is true for a garbage node `Z` if the term `(PURE Z M NODES SONS L)` is satisfied at some point during an appending phase:

Eventuality 3

```
    (OR (ACCESSIBLE Z M NODES ROOTS SONS)
        (AND (NUMBERP Z) (LESSP Z NODES)
             (MEMBER CHI '(CHI7 CHI8))
             (PURE Z M NODES SONS L)))
→ ◇(OR (ACCESSIBLE Z M NODES ROOTS SONS)
        (AND (NUMBERP Z) (LESSP Z NODES)
             (MEMBER CHI '(CHI0 CHI1 CHI2 CHI3 CHI4 CHI5 CHI6))
             (PURE Z M NODES SONS NODES)))
```

It must be shown that if, with respect to some state, if `Z` is a garbage node and

```
        (AND (NUMBERP Z) (LESSP Z NODES)
             (MEMBER CHI '(CHI7 CHI8))
             (PURE Z M NODES SONS L))
```

is satisfied, then this term continues to be satisfied until the conclusion of Eventuality 3 becomes true. As in the proof of Eventuality 2, it is clear that the value of the term is unaffected by the mutator, and we may restrict our attention to the arcs of the collector.

The arc from `CHI7` to `CHI0` clearly renders the conclusion true, and there is no transformation associated with the arc from `CHI7` to `CHI8`. Hence, we need only consider the two arcs from `CHI8` to `CHI7`. If `(COLOR L M)` is initially true, then only a whitening operation is performed, which cannot cause `Z` to become impure. The remaining case, in which the garbage node `L` is appended to the free list, is covered by Constraint 3. The proof of Eventuality 3 is completed with the same measure and argument that were used to prove Eventuality 1.

The last link in our chain of eventualities states that every node eventually satisfies the hypothesis of Eventuality 3:

Eventuality 4

```
    (AND (NUMBERP Z) (LESSP Z NODES))
→ ◇(OR (ACCESSIBLE Z M NODES ROOTS SONS)
        (AND (NUMBERP Z) (LESSP Z NODES)
             (MEMBER CHI '(CHI7 CHI8))
             (PURE Z M NODES SONS L)))
```

Since the value of `(AND (NUMBERP Z) (LESSP Z NODES)` is not affected by any transition, we need only produce an appropriate measure. We obtain this measure as a modification of the measure that was used for Eventualities 1 and 3:

```
  (LEX (LIST (PHASE CHI '((CHI6 CHI5 CHI4 CHI3 CHI2 CHI1 CHI0)
                          (CHI8 CHI7)))
             (DIFFERENCE NODES L)
             (LOC CHI '(CHI8 CHI7))
             (DIFFERENCE NODES OBC)
             (PHASE CHI '((CHI6 CHI5 CHI4)
                          (CHI3 CHI2 CHI1)
                          (CHI0)))
             (DIFFERENCE NODES H)
             (LOC CHI '(CHI6 CHI5 CHI4))
             (DIFFERENCE NODES I)
             (LOC CHI '(CHI3 CHI2 CHI1))
             (DIFFERENCE SONS J)
             (DIFFERENCE ROOTS K)))
```

Suppose that the hypothesis of Eventuality 4 is satisfied and the conclusion is not. As long as `CHI` is in the appending phase, it is clear from consideration of the first three components of the above measure that its value decreases until the marking phase is entered. Once we are in the marking phase, the value of the measure continues to decrease, as in the proof of Eventuality 2, until the next appending phase is entered. But upon entering the appending phase, `L` = `0` and the conclusion of Eventuality 4 is trivially satisfied.

Finally, having proved these four eventualities, the `PROVE-CHAIN` macro may be applied to yield the desired result:

Eventuality 5

```
    (AND (NUMBERP Z) (LESSP Z NODES))
→ ◇(ACCESSIBLE Z M NODES ROOTS SONS)
```

6 Conclusion

Undeniably, mechanical program verification requires careful attention to detail. But while this may be quite tedious, it need not be prohibitive, since at the lowest level, the details are managed automatically by the prover. On the other hand, the consequences of carelessness, which is often unavoidable in less formal approaches, may be devastating.

Thus, our proof of correctness of Ben-Ari's algorithm required proving twenty-two program invariants, each of which involves checking various program transitions. Over one hundred lemmas characterizing the behavior of relevant functions were required for these invariance proofs. (Many of these lemmas were established prior to the development of the correctness proof, while some were added later as needed.) However, once the required lemmas and invariants were identified, all of their proofs were constructed mechanically, without assistance from the user. Our certainty of the final result now depends only on our trust in the theorem prover, without our having to examine any of the proofs.

But increased confidence in results is not the only benefit of mechanical proof systems. As illustrated by the example at hand, an important aspect of the process of generating a mechanical proof is the explication of all implicit assumptions that are relevant to the proof. In this case, we found it necessary to identify the essential properties of the collector's appending operation. These properties would be required of any implementation, but were ignored in all previously published informal proofs. We found the results of this exercise to be somewhat counterintuitive, with surprising implications concerning the structure of our proof.

The mechanical proof process also automatically produces a complete and precise record of the dependencies among assumptions, intermediate lemmas, and final results. Tracing these dependencies provides insight into the structure of a proof that may be relevant to proposed variations of the algorithm or modification of the assumptions. In our example, we found that our original proof, which was based faithfully on Ben-Ari's model, made no use of two of his basic assumptions: one concerning the initial state of the data structure, according to which all nodes are linked together on the free list, and another guaranteeing the accessibility of the source node of the mutation instruction. We were then able to discard both of these assumptions without affecting the proof.

A more startling conclusion drawn from our proof is the observation that Invariant 21, which states the persistent accessibility of the target node `Q` of the mutation instruction, actually depends on Invariant 20, the msin safety property of the program. While this property of `Q` was taken as obvious and either used implicitly or stated without proof in [Ben84], [DLM78], [Pix88], and [Van87], the complexity of its proof could only be revealed after the assumptions concerning the appending operation were made explicit. A corollary of this observation is the dependence of the liveness result, Eventuality 5, on the safety result, another relationship that previously went unnoticed.

Another point revealed by our proof is that the correctness of this program does not depend on the full assumption of fairness, which guarantees that no process ever permanently ceases to make progress. In this case, the helper process used in the proof of each liveness property is always the collector and never the mutator. It follows (although this observation cannot be expressed in our formal notation) that all of our results remain valid even if the mutator is allowed to halt at any point during an execution. In particular, the mutator need not color the new target of a pointer after redirecting it, as long as it does perform any further mutation.

In spite of the complexity of this problem, the algorithm is simple enough to be coded easily in a very primitive language for which mechanical verification is feasible. It remains true, however, that in contrast, real programming languages are generally semantically vague. The need for expressiveness or efficient implementation often motivates the design of language constructs and data types that are difficult to formalize. Some progress has been made in this area [Ram89,Sut90], but it remains a challenge to future research.

References

[Ben84] Ben-Ari, M., *Algorithms for On-the-Fly Garbage Collection*, ACM Toplas 6, July 1984.

[BGK89] Boyer, R. S., Goldschlag, D. M., Kaufmann, M., and Moore, J, *Functional Instantiation in First Order Logic*, Tech. Report 44, Computational Logic, Inc., Austin, TX, 1989.

[BoM79] Boyer, R. S. and Moore, J S., A Computational Logic, Academic Press, New York, 1979.

[BoM88] Boyer, R. S. and Moore, J, A Computational Logic Handbook, Academic Press, Boston, 1988.

[DLM78] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M., *On-the-Fly Garbage Collection: An Exercise in Cooperation*, ACM 21 (11), November 1978.

[MaP81] Manna, Z. and Pnueli, A., *Verification of Concurrent Programs: the Temporal Framework*, in The Correctness Problem in Computer Science, edited by Boyer, R. S. and Moore, J, Academic Press, London, 1981.

[MaP84] Manna, Z. and Pnueli, A., *Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs*, Science of Computer Programming 4 (1984), North-Holland.

[Pix88] Pixley, C., *An Incremental Garbage Collection Algorithm for Multi-mutator Systems*, Distributed Computing (3), 1988.

[Ram89] Ramsey, N., *Developing Formally Verified Ada Programs*, Proceeding of the Fifth International Conference on Software Specification and Design, May 1989.

[Rus90] Russinoff, D. M., *Verifying Concurrent Programs with the Boyer-Moore Prover*, Tech. Report STP/ACT-218-90, MCC, Austin, TX, 1990.

[Rus92] Russinoff, D. M., *A Verification System for Concurrent Programs Based on the Boyer-Moore Prover*, to appear in Formal Aspects of Computing.

[Ste84] Steele, G. L., Common LISP: The Language, Digital Press, Burlington, MA, 1984.

[Sut90] Sutherland, I., *Formal Verification of Mathematical Software*, Tech. Report RADC-TR-90-53, Odyssey Research Assoc., Inc., May 1990.

[Van87] Van de Snepscheut, J. L. A., *"Algorithms for On-the-Fly Garbage Collection" Revisited*, Information Processing Letters (24), March 1987.