A Foundation of Parallel Programming

Jayadev Misra*
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
(512) 471-9547
misra@ratliff.cs.utexas.edu

1 Introduction

This monograph introduces a programming theory (called UNITY) that is applicable to the design of parallel (concurrent/distributed/multi-process) programs. This theory includes a very simple computational model and a logic that is appropriate for specifying and reasoning about such programs. The computational model was first proposed in Chandy []; a full account of this work appears in Chandy and Misra []. This manuscript contains an abbreviated version of the theory and a few small examples to illustrate the theory.

In Section 2, we give a brief description of the computational model and a notaion for writing programs; this material is included in Chapters 1 and 2 of []. In Section 3, we show how this model addresses some of the issues of programming, and of parallel programming, in particular. Section 4 contains a description of the logic and some of the inference rules that are used in this monograph; most of this material, though not all, is in Chapter 3 of []. The next three sections illustrate some typical applications of the theory. Section 5 contains a specification and a solution of the "termination detection problem," a problem of some interest in implementing message communicating systems. Section 6 contains a simplified version of a problem dealing with resource allocations. Section 7 gives a specification and an implementation of a buffer program that is to be interposed between a producer and a consumer. The final section suggests some research directions.

2 The Computational Model and A Programming Notation

A UNITY program consists of a declaration of variables, their initial values, and a set of statements. In this monograph, the types of variables will be limited to integers and booleans, and the data structures to fixed size arrays; however, the programming theory is not based upon any particular data type. Initial values need not be specified for all variables; variables without a specified initial value have arbitrary values (consistent with their types) initially. Each statement in a program is an assignment statement that assigns values to one or more variables.

*This work was partially supported by ONR Contracts N00014-86-K-0763 and N00014-87-K-0510 and by a grant from the John Simon Guggenheim Foundation.

An execution of a program consists of an infinite number of steps. It begins from any state satisfying the prescribed initial conditions. In each step of the execution some statement is selected, nondeterministically, and executed. The only constraint on the nondeterministic selection of statements is that every statement is selected infinitely often; this is called the "fairness' rule.

Now we briefly describe a notation for expressing UNITY programs. This description is informal; see Chapter 2 of [] for a formal description of the syntax.

A program consists of several *sections*: *declare-section* for declarations of variables, *initially-section* for prescribing initial values of variables, and *assign-section* to list the statements. (Another section, called the *always-section*, is not used in this monograph.) We do not describe the syntax of the *declare-section* which is similar to the syntax used for variable declarations in PASCAL, nor the *initially-section* whose syntax closely resembles that of the *assign-section*.

The statements in an *assign-section* are separated by the symbol $[\!]$. The following *assign-section* consists of two statements

$$\begin{array}{ll} & x \;:=\; y \\ [\!] & y \;:=\; x \end{array}$$

The statements of a program may also be defined using *quantification*; the statements are obtained by instantiating a generic statement, as in

$\langle [\!]\; i \;:\; 0 \le i \le 2 \;\;::\;\; A[i] \;:=\; B[i] \rangle$

This represents a set of three statements, one for each value of i in the range $0 \le i \le 2$. The statements obtained by substituting each possible value of i in $A[i] := B[i]$ are,

$A[0] \;:=\; B[0] \;\;[\!]\;\; A[1] \;:=\; B[1] \;\;[\!]\;\; A[2] \;:=\; B[2]$

This quantification mechanism is so useful that we employ it in many different contexts. For instance

$\langle \wedge\; i \;:\; 0 \le i \le N \;\;::\;\; B[i] \rangle$

stands for $B[0] \wedge B[1] \wedge \ldots \wedge B[N]$. The operation—$\wedge$ in this case— is always commutative and associative, and it follows the opening bracket; the bound variables—i in this case— come next (they are always of integer type); the boolean expression appearing within ":" and "::" specifies the possible values of the bound variables (the number of possible values must be finite); finally, the expression or the syntactic unit that is to be instantiated is given. The boolean expression defining the range of the bound variables is sometimes omitted when the range is defined in the accompanying text. (If there are no values of the bound variables satisfying the boolean expression then the instantiation for statement yields an empty statement, and for $\wedge, \vee$ yield *true, false*, respectively.)

The individual assignment statements in an *assign-section* are, in general, multiple assignments as in

$x, y \;:=\; y, x$

An alternative way of writing the above, to denote that x, y are assigned their values in parallel, is

$x \;:=\; y \;\;\|\;\; y \;:=\; x$

We may employ quantification within a statement as in

$\langle \|\; i \;:\; 0 \le i \le N \;\;::\;\; A[i] \;:=\; B[i] \rangle$

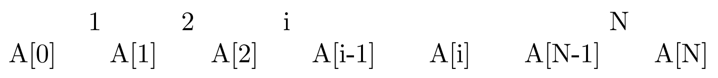

Figure 1: A Shift Register with N Elements

which assigns elements of array B to the corresponding elements of array A in parallel (i.e., in one step).

We also employ conditional expressions for assignments and separate them by $\sim$ as in,

$$\begin{array}{lllll} x & := & -1 & \text{if} & y < 0 \quad \sim \\ & & 0 & \text{if} & y = 0 \quad \sim \\ & & 1 & \text{if} & y > 0 \end{array}$$

This denotes that x is assigned either $-1, 0$ or 1 based on whether y is negative, zero, or positive.

All statements are deterministic: if two different conditions in a conditional expression are satisfied then they yield the same value for the assignment. Furthermore, each statement execution terminates in each program state; this is easily guaranteed for assignment statements in which the function calls, if any, are guaranteed to terminate.

Notation: $\forall, \exists$ are used synonymously with $\wedge, \vee$, respectively. ∇

3 Features of the Computational Model

In this section we show (very briefly) how the proposed computational model is adequate for representing a number of programming constructs and why it is useful for program refinement and structuring. We will also illustrate these points in reference to the examples in Sections 5,6 and 7.

3.1 Representations of Programming Constructs

Synchrony

A basic construct in parallel programming is the synchronous execution of two (or more) actions. The multiple-assignment statement captures synchrony. For instance,

$$x, y \;:=\; 3, 5$$

prescribes that assignments to x and y must be performed in an "atomic" manner. In UNITY, we deliberately ignore the agent which performs the assignment; thus if x, y are accessible to one machine then the assignment may be performed relatively simply whereas if these are local variables of two different machines then some synchronization mechanism may have to be invoked. Abstracting away from the implementation details has the advantage that a relatively simple mechanisms—multiple-assignment—may always be employed to represent synchronous executions.

As an example, consider a simple shift register that is shown schematically in Figure 1. The shift register has N elements; the input,output lines of the i^{th} element are $A[i-1]$ and $A[i]$, respectively, for all i, $1 \leq i \leq N$. In each step, all elements transfer the data from their input lines to their output lines. This operation is represented by

$$\langle \| \; i \;:\; 1 \leq i \leq N \;\;::\;\; A[i] \;:=\; A[i-1] \rangle$$

Asynchrony

Another basic construct in parallel programming is to sepcify that two (or more) actions are executed in arbitrary order. The statements of a UNITY program are executed nondeterministically, and nondeterminism captures the essence of asynchrony. Again, the machines on which the actions are executed are irrelevant; if the actions are on one machine the the UNITY program may be implemented in a straightforward fashion and if the actions are to be executed by multiple machines some form of communication among machines may have to be employed to ensure appropriate access to shared data. It is best to ignore these implementation issues during the higher levels of program design.

The following program computes the maximum, m, of an array A, of N integer elements (A is not declared below).

```
program maximum
    declare    $m$ : integer
    initially  $m = -\infty$
    assign     $\langle [\!] \; i \; : \; 0 \leq i < N \;\; :: \;\; m \; := \; \max(m, A[i]) \rangle$
end {maximum}
```

The typical sequential program for finding the maximum prescribes an order in which the elements of the array A are to be scanned. We can execute the above program on a sequential computer by prescribing an order for statement executions; for instance, we may prescribe a round-robin schedule in which the statement to be executed following the i^{th} statement—i^{th} statement is $m := \max(m, A[i])$—is the statement $(i + 1)$ mod N, and initially the 0^{th} statement is executed.

We can also partition the statements of this program for parallel execution by multiple processors that have access to a common memory in which m resides. Also, we can implement this program on a message passing architecture; the details are given below. The point is that a UNITY program offers a variety of options for implementations on different architectures. The number of implementation options are limited had we started with a more deterministic program, for instance, in which the order of statement executions was prescribed.

Synchrony and Asynchrony

The following little program illustrates the use of both synchrony and asynchrony. The programming task is to sort an array of integers A, $A[0..N]$ in ascending order. The strategy employed is to pick any pair of adjacent elements and exchange them if they are out of order. The reader may convince himself that eventually A will be sorted in ascending order (in the absence of a logic, we cannot even state this fact, let alone prove it).

```
program sort
    assign  $\langle [\!] \; i \; : \; 0 \leq i < N \;\; ::$
              $A[i], A[i+1] \; := \; \min(A[i], A[i+1]), \max(A[i], A[i+1])$
            $\rangle$
end {sort}
```

There are N statements in this program, corresponding to each value of i in the range $0 \leq i < N$, and their execution order is nondeterministic. Each statement execution, however, requires synchronous assignments of the appropriate values to $A[i]$ and $A[i + 1]$.

Termination

Consider a state of a program in which execution of any statement causes no state change; we call such a state a *fixed point.* A program may have no fixed point (i.e., a program that is expected to produce an unending sequence of numbers). However if a program is at a fixed point, all its future states can be predicted, and hence, an implementation may decide to halt the execution of the program on a machine (or machines). Reaching of a fixed point is typically termed "termination" in sequential programming. We view termination as a feature of an implementation. The study of fixed points is, however, important to demonstrate that some programs are guaranteed to reach fixed points, and that some programs are guaranteed never to reach any fixed point (e.g., freedom from deadlock). The detection of a fixed point during program execution is, in general, a nontrivial task, particularly if the program is implemented on asynchronous processors; the problem is closely related to termination detection of Section 5.

Multiprocess Programs

There is no notion of a process in UNITY; however, the variables and the statements of a UNITY program may be partitioned for execution among a set of processes. Any fair interleaving of the fair executions of the partitions is an execution of the original program. Thus any implementation of "interleaved semantics"—i.e., where executions of any two actions by different processes is equivalent to their execution in some order—also provides an implementation of a UNITY program as a set of processes. Grain of atomicity in interleaving depends on the nature of the variables that are shared among the processes, and how they are accessed and updated; for details see Chapter 4 of [].

As an example consider the program *maximum* given earlier. Suppose that the N variables—$A[0]$,$A[1] \ldots A[N-1]$—are partitioned among N processes, the i^{th} process having $A[i]$ as a local variable. The variable m is in some shared memory that can be accessed by all processes. The statements are also partitioned among these processes, the i^{th} process executing

$$m \ := \ \max(m, A[i])$$

The processes may then execute their codes independently as long as their accesses to m are mutually exclusive.

A strategy for implementing any shared-variable multiprocess program in a message passing architecture is to circulate a token among all the processes; the token carries the values of all the shared variables, and only the token holder may examine and/or update the shared variables. Using this strategy, program *maximum* may be implemented by introducing a token that carries the value of m.

A UNITY program can be efficiently implemented on a message passing architecture provided its variables and statements can be partitioned in such a manner that the shared variables (i.e., those that appear in statements of two partitions) are of type sequence (representing a channel), each shared variable is shared by exactly two partitions (representing that a channel is directed from exactly one sender to exactly one receiver), and the only operations on the shared variables are (1) appending to the end of the sequence in one partition (corresponding to sending a message), and (2) removing the head item of a sequence (corresponding to receiving a message) provided the sequence is nonempty. A theory of message communication appears in Chapter 8 of []. It is not difficult to see that processes communicating in other fashions—rendezvous-style communication, broadcast, wait and signal on a condition, etc.—also have succinct representations within UNITY.

3.2 Program Refinements

A UNITY program is not tied to any particular architecture. It seems feasible therefore to develop a program in a series of refinement steps; the higher level concerns are about the problem being solved and the lower level concerns are about the architecture on which the problem is to be solved. It is important to note that all the refinements, except the very last step, can be carried out entirely within the UNITY notation, from a high level algorithm description to the very lowest level representation. The very last step of refinement maps a UNITY program to a particular architecture and/or specifies the order in which the statements are to be executed. For instance, when mapping a UNITY program to a sequential machine we must prescribe an order in which the statements are to be executed and guarantee that this order obeys the fairness rule. The mappings of UNITY programs to architectures is outside the UNITY theory. We will describe only informally how specific programs are mapped for efficient executions on specific architectures; see Chapter 4 of [] for a more elaborate discussion.

At present it seems more attractive to refine specifications rather than the program code. The reason is that proving the correctness of a specification refinement is easier.

3.3 Correctness and Complexity

A traditional sequential program can be analyzed to establish its correctness and also to obtain a measure of its efficiency when executed on a traditional (von Neumann) computer. In this sense such a program embodies the logical steps of the algorithm as well as information about how these steps must be orchestrated. (One consequence of this is that correctness proofs are often difficult for programs which include several "optimization tricks.") The issues of correctness and complexity are totally separate in UNITY. The correctness of a UNITY program (with respect to a given specification) is a well-defined question. However, the complexity of a UNITY program is *not* well-defined; it is meaningful only after a mapping of the program to a specific architecture (and a specific execution schedule) is prescribed.

3.4 Program Structuring

Processes have a dual role in parallel programming. They are the units of structuring—i.e., a program may be understood (specified, verified, and developed) process by process. Additionally, they are the implementation units, i.e., a process (or a set of processes) are mapped to a physical processor. In many cases the best way to understand or develop a program does not lead directly to the most efficient implementation. UNITY provides a framework for separating these two issues. For instance, consider a program H consisting of two component programs, F and G—the set of statements of H is the union of the statements of F and G (and other conditions apply to *declare*- and *initially-sections*; see Section 4 for the definition of *union*). It may be best to understand H in terms of F and G and then implement H by partitioning its statements in an entirely different manner into F', G' say. Each of F', G' may contain statements from both F and G. This view of programming is useful when a computation can be structured as a set of tasks (F, G, for instance) and each task is executed by a cooperating set of processes (F', G', for instance). Section 5 contains an example where it is best to understand a computation as an interleaved execution of two tasks, each task being executed on a fixed set of processors.

4 A Brief Introduction to UNITY Logic

The structure of a UNITY program dictates that we associate properties with a program, not with points in a program because UNITY programs have no textual program points. Thus, we write "I is invariant in F" to denote that predicate I holds at all times during execution of program F. Associating properties with programs has the advantage that properties of compositions of programs can then be derived somewhat more easily.

The kinds of properties that we shall be dealing with can be broadly divided into two classes: safety and progress. Safety properties establish taht "nothing bad ever happens," and progress properties establish that "something good happens eventually." In addition, we will introduce a predicate to deal with fixed points of a program.

We introduce three relations—*unless*, *ensures*, *leads-to*—each of which is a binary relation on a pair of predicates. (Each predicate may name program variables, bound variables, and free variables as well as constants, functions, and relations.) The safety properties are written using *unless* (and the initial conditions); the progress properties are written using *ensures* or *leads-to*.

Safety properties can be established by applying induction on the length of computations. Progress properties, however, are more difficult to establish, because they often rely critically on the fairness assumption. For instance, consider a program that has two statements, one that decreases variable x by 1 and the other that decreases y by 1. We can assert both x, y will become arbitrarily small in a sufficiently long computation. (This property does not hold in the absence of a fairness assumption.) The assertion cannot be proven by applying induction on the length of computations, because there is no "variant" function that decreases with each step of the computation. We develop a logic in which the fairness assumption is captured within one relation (*ensures*) and we define a second relation (*leads-to*) inductively, using *ensures* as the basis of induction.

Convention: A property having free variables is assumed to be universally quantified over all possible values of the free variables. Thus,

$$x = k \quad \textit{unless} \quad x > k$$

where k is free, stands for

$$\langle \forall\ k\ ::\ x = k \quad \textit{unless} \quad x > k \rangle. \qquad \bigtriangledown$$

In the following description, all properties refer to a single program; the program name is omitted for the sake of brevity.

Notation: We use p, q, r, b (with or without subscripts and primes) to denote predicates. Symbol s denotes a statement of a program.

We use

$$\{p\} \quad s \quad \{q\}$$

to denote that if s is executed in a state satisfying p then q holds upon completion of s. We assume that every statement terminates in every state; this is guaranteed because we limit ourselves to assignment statements in which only total functions may be used.

unless

For a given program

$$p \quad \textit{unless} \quad q$$

denotes that once predicate p is true it remains true at least as long as q is not true. Formally (using Hoare's notation)

$$p\ \textit{unless}\ q \quad \equiv \quad \langle \text{for all statements } s \text{ of the program} :: \ \{p \wedge \neg q\} \quad s \quad \{p \vee q\} \rangle$$

i.e., if $p \wedge \neg q$ holds prior to execution of any statement of the program then $p \vee q$ holds following the execution of the statement.

It follows from this definition that if p holds and q does not hold in a program state then in the next state either q holds, or $\wedge\neg q$ holds; hence, by induction on the number of statement executions, $p \wedge \neg q$ continues to hold as long as q does not hold. Note that it is possible for $p \wedge \neg q$ to hold forever. Also note that if $\neg p \vee q$ holds in a state then $punlessq$ does not tell us anything about future states.

Examples:

1. Integer variable x does not decrease.

   $x = k$ *unless* $x > k$
   or $x \geq k$ *unless false*

2. A message is received (i.e., predicate *rcvd* holds) only if it had been went earlier (i.e., predicate *sent* already holds).

   $\neg$*rcvd unless sent*

3. Variable x remains positive as long as y is.

   $x > 0$ *unless* $y \leq 0$

4. x, y change only synchronously.

   $x = m \ \wedge \ y = n$ *unless* $x \neq m \ \wedge \ y \neq n$

5. x, y never change synchronously

   $x = m \ \wedge \ y = n$ *unless* $(x = m \ \wedge \ y \neq n) \ \vee \ (x \neq m \ \wedge \ y = n)$ $\bigtriangledown$

An interesting application of *unless* is in defining auxiliary (or history) variables. Traditionally, auxiliary variables are defined by augmenting the code of a given program. A preferable way is to state a safety property (using *unless*) that describes the relationship between the auxiliary and the other variables. As an example let s denote an integer variable and let $\bar{s}$ be an auxiliary variable whose value is the number of times that s has been increased. We may define $\bar{s}$ as follows.

initially $\bar{s} = 0$
$\bar{s} = m \ \wedge \ s \leq n$ *unless* $\bar{s} = m + 1 \ \wedge \ s > n$

The above *unless* property may be read informally as "as long as s does not increase $(s \leq n)$ $\bar{s}$ retains its value; when s increases $(s > n$, $\bar{s}$ is increased by 1."

Stable, Invariant

Two special cases of *unless*—stable, invariant—are of importance

p is stable $\equiv$ p *unless false*
p is invariant $\equiv$ (initially p) $\wedge$ (p is stable)

¿From the definition, p is stable denotes that for all statements s (of the given program)

$\{p\}\ \ s\ \ \{p\}$

Thus p remains true once it is true. An invariant is initially true and remains true throughout any execution of the program.

Derived Rules for unless

The following derived rules are extensively used in proofs. Most of these are stated and proved in Chapter 3 of [].

- Reflexivity

$$p \ \mathit{unless}\ p$$

- Antireflexivity

$$p \ \mathit{unless}\ \neg p$$

- consequence weakening

$$\frac{p\ \mathit{unless}\ q\ ,\ q\ \Rightarrow\ r}{p\ \mathit{unless}\ q}$$

Corollary:

$$\frac{p\ \Rightarrow\ q}{p\ \mathit{unless}\ q}$$

- conjunction and disjunction

$$\frac{p\ \mathit{unless}\ q\ ,\ p'\ \mathit{unless}\ q'}{\begin{array}{llll} p\ \wedge\ p' & \mathit{unless} & (p \wedge q')\ \vee\ (p' \wedge q)\ \vee\ (q \wedge q') & \{\text{conjunction}\}\ , \\ p\ \vee\ p' & \mathit{unless} & (\neg p \wedge q')\ \vee\ (\neg p' \wedge q)\ \vee\ (q \wedge q') & \{\text{disjunction}\} \end{array}}$$

Simpler forms of conjunction and disjunction are often useful; these are obtained from the above rule by weakening the consequence to $q \vee q'$ in both cases:

- (simple conjunction and simple disjunction)

$$\frac{p\ \mathit{unless}\ q\ ,\ p'\ \mathit{unless}\ q'}{\begin{array}{llll} p\ \wedge\ p' & \mathit{unless} & q\ \vee\ q' & \{\text{simple conjunction}\} \\ p\ \vee\ p' & \mathit{unless} & q\ \vee\ q' & \{\text{simple disjunction}\} \end{array}}$$

A corollary of the simple conjunction and disjunction rules is, if I, J are stable (invariant) then so are $I \wedge J, I \vee J$.

The following rule generalizes the conjunction and disjunction rules to an arbitrary number—perhaps infinite—of *unless*es. For proof, see []. In the following, m is quantified over some arbitrary set.

(general conjunction and general disjunction)

$$\frac{\langle \forall\ m\ ::\ p.m\ \mathit{unless}\ q.m \rangle}{\begin{array}{llll} \langle \forall\ m\ ::\ p.m\rangle\ \ \mathit{unless}\ \ \langle \forall\ m\ ::\ p.m \vee q.m\rangle & \wedge & \langle \exists\ m\ ::\ q.m\rangle & \{\text{general conjunction}\} \\ \langle \exists\ m\ ::\ p.m\rangle\ \ \mathit{unless}\ \ \langle \forall\ m\ ::\ \neg p.m \vee q.m\rangle & \wedge & \langle \exists\ m\ ::\ q.m\rangle & \{\text{general disjuction}\} \end{array}}$$

Corollary: Let M by any total function over program states. Variables m is free in the following rule.

$$\frac{p\ \wedge\ M = m\ \mathit{unless}\ (p\ \wedge\ M \neq m)\ \vee\ q}{p\ \mathit{unless}\ q}$$

Corollary (free variable elimination)

Let x be a set of program variables and k be free in the following.

$$\frac{p\ \wedge\ x = k\ \mathit{unless}\ q}{p\ \mathit{unless}\ q}$$

ensures

For a given program,

$p\ \mathit{ensures}\ q$

implies that $p\ \mathit{unless}\ q$ holds for the program and if p holds at any point in the execution of the program then q holds eventually. Formally,

$p\ \mathit{ensures}\ q\ \equiv\ p\ \mathit{unless}\ q\ \wedge\ \langle \exists\ \text{statement}\ s :: \{p \wedge \neg q\}\ \ s\ \ \{q\}\rangle$.

It follows from this definition that once p is true it remains true at least as long as q is not true (from $p\ \mathit{unless}\ q$). Furthermore, from the rules of program execution, statement s will be executed sometime after p becomes true. If q is still false prior to the execution of s then $p \wedge \neg q$ holds and the execution of s then establishes q.

Example 1: Consider a program with the following *assign-section* (where x, y are integer variables).

$x\ :=\ x = q\ \ [\!]\ \ y\ :=\ y - 1$

To show that
$x = k\ \mathit{ensures}\ x < k$
We first prove
$x = k\ \mathit{unless}\ x < k$
i.e., $\{x = k\}\ \ x\ :=\ x - 1\ \ \{x = k\ \vee\ x < k\}$ and,
$\{x = k\}\ \ y\ :=\ y - 1\ \ \{x = k\ \vee\ x < k\}$
Then we prove
$\{x = k\}\ \ x\ :=\ x - 1\ \ \{x < k\}$ $\bigtriangledown$

A Note on Quantification

In order to prove $x = k\ \mathit{ensures}\ x < k$ we have to show

$\langle \forall\ k\ ::\ \langle \exists\ s\ ::\ \{x = k\}\ \ s\{x < k\}\rangle\rangle$

not

$\langle \exists\ s\ ::\ \langle \forall\ k\ ::\ \{x = k\}\ \ s\{x < k\}\rangle\rangle$ $\bigtriangledown$

Derived Rules for ensures

We show two commonly used rules; for a longer list of rules see [].

- Reflexivity

$$p\ \textit{ensures}\ p$$

- Consequence Weakening

$$\frac{p\ \textit{ensures}\ q\ ,\ q\ \Rightarrow\ r}{p\ \textit{ensures}\ r}$$

Corollary:

$$\frac{p\ \Rightarrow\ q}{p\ \textit{ensures}\ q}$$

leads-to

For a given program $p\ leads-to\ q$, abbreviated as $p \mapsto q$, denotes that once p is true, q is or becomes true. Unlike *ensures*, p may not remain true until q becomes true. The relation *leads-to* is defined inductively by the following rules. The first rule is the basis for the inductive definition of *leads-to.* The second rules states that *leads-to* is a transitive relation. In the third rule, $p.m$, for different m, denote a set of predicates. This rule states that if every predicate in a set *leads-to* q then their disjunction also *leads-to* q.

(basis)

$$\frac{p\ \textit{ensures}\ q}{p\ \mapsto\ q}$$

(transitivity)

$$\frac{p\ \mapsto\ q\ ,\ q\ \mapsto\ r}{p\ \mapsto\ r}$$

(disjunction)

$$\frac{\langle \forall\ m\ ::\ p.m\ \mapsto\ q\rangle}{\langle \exists\ m\ ::\ p.m\rangle\ \mapsto\ q}$$

Example 2: For the program in Example 1, we show that $\langle \forall\ k\ ::\ x \geq k\ \mapsto\ x < k\rangle$

$\langle \forall\ k, r\ :\ r \geq 0\ ::\ x = k + r\ \mapsto\ x < k\rangle$, shown below
$\langle \forall\ k\ ::\ \langle \forall\ r\ :\ r \geq 0\ ::\ x = k + r\ \mapsto\ x < k\rangle$, rewriting the above
$\langle \forall\ k\ ::\ \langle \exists\ r\ :\ r \geq 0\ ::\ x = k + r\rangle\ \mapsto\ x < k\rangle$, disjunction on the above
$\langle \forall\ k\ ::\ x \geq k \mapsto\ x < k\rangle$, rewriting the above

Proof of $\langle \forall\ k, r\ :\ r \geq 0\ ::\ x = k + r\ \mapsto\ x < k\rangle$

The proof is by induction on r.

$r = 0:$ Proof of $\langle \forall\ k\ ::\ x = k\ \mapsto\ x < k \rangle$

$\langle \forall\ k\ ::\ x = k\ \textit{ensures}\ x < k \rangle$, from Example 1
$\langle \forall\ k\ ::\ x = k\ \mapsto\ x < k \rangle$, using the definition of $\mapsto$

$r > 0:$

$\langle \forall\ k\ ::\ x = k\ \mapsto\ x < k \rangle$, proven above
$\langle \forall\ k\ ::\ x < k\ \mapsto\ x < k \rangle$, property of $\mapsto$ (see Section ??)
$\langle \forall\ k\ ::\ x < k + q\ \mapsto\ x < k \rangle$, disjunction on the above two
$\langle \forall\ k\ ::\ x = k + r\ \mapsto\ x < k + 1 \rangle$, induction hypothesis
$\langle \forall\ k\ ::\ x = k + r\ \mapsto\ x < k \rangle$, transitivity on the above two $\bigtriangledown$

Derived Rules for leads-to

As Example 2 shows, proofs can be very long, even for trivial facts, if they are constructed starting from the definitions. The derived rules for *leads-to* are particularly effective in constructing succinct proofs.

- implication

$$\frac{p\ \Rightarrow\ q}{p\ \mapsto\ q}$$

- impossibility

$$\frac{p\ \mapsto\ \textit{false}}{\neg p}$$

- general disjunction: In the following m is quantified over an arbitrary set.

$$\frac{\langle \forall\ m\ ::\ p.m\ \mapsto\ q.m \rangle}{\langle \exists\ m\ ::\ p.m \rangle\ \mapsto\ \langle \exists\ m\ ::\ q.m \rangle}$$

- PSP

$$\frac{p\ \mapsto\ q\ ,\ r\ \textit{unless}\ b}{p\ \wedge\ r\ \mapsto\ (q \wedge r)\ \vee\ b}$$

- Completion: In the following m is quantified over a finite set.

$$\frac{\begin{array}{c}\langle \forall\ m\ ::\ p.m\ \mapsto\ q.m \rangle\ ,\\ \langle \forall\ m\ ::\ q.m\ \textit{unless}\ b \rangle\end{array}}{\langle \forall\ m\ ::\ p.m \rangle\ \mapsto\ \langle \forall\ m\ ::\ q.m \rangle\ \vee\ b}$$

- Induction: In the following M is a total function mapping program states to a well-founded set $(W, \prec)$.

$$\frac{\langle \forall\ m\ :\ m \in W\ ::\ p\ \wedge\ M = m\ \mapsto\ (p \wedge M \prec m)\ \vee\ q \rangle}{p\ \mapsto\ q}$$

We now rework Example 2 using these derived rules to obtain a succcinct proof. In particular, use of explicit induction in the proof is avoided by applying the induction rule.

Note: The mapping function M need be defrined only over the program states satisfying $p \wedge \neg q$.

$$\begin{array}{ll}
x = m \textit{ ensures } x < m & \text{, from Example 1} \\
x = m \;\mapsto\; x < m & \text{, from the definition of } \mapsto \\
x \geq k \textit{ unless } x < k & \text{, antireflexivity of } \textit{unless} \\
x \geq k \;\wedge\; x = m \;\mapsto\; (x \geq k \wedge x < m) \;\vee\; x < k & \text{, PSP rule on the above two} \\
x \geq k \;\mapsto\; x < k & \text{, induction rule (see below)} \qquad \bigtriangledown
\end{array}$$

In order to apply the induction rule we postulate a mapping function M defined over the states that satisfy $x \geq k$ (i.e., $p \wedge \neg q$); the mapping function merely has the value of x; the well-founded set consists of $\{n|n \geq k\}$ that is ordered according to the usual ordering of integers.

Fixed Point

A fixed point of a program is a state that does not change, i.e., execution of a statement has no effect in this state. Clearly then every statement's left and right sides are equal in this state. Using this observation we define a predicate *FP* for a program which characterizes all and only fixed points. *FP* is the conjunction of all predicates that are obtained by equating the left and the right sides of each statement.

Example 4: We compute fixed points of a few programs whose *assign-sections* are shown below.

1. $x, y := y, x$
   $FP \;\equiv\; (x = y)$

2. $k \;\; := k + 1$
   $FP \;\equiv\; k = k + 1$
   $\quad\;\; \equiv\; \textit{false}$

3. $k \;\; := k + 1 \qquad \text{if} \quad k < N$
   $FP \;\equiv\; k < N \;\Rightarrow\; k = k + 1$
   $\quad\;\; \equiv\; k \geq N$

4. $\langle [\!] \; i \;:\; 0 \leq i < N \;::\; m \;:=\; \max(m, A[i]) \rangle$
   FP
   $\equiv\; \langle \wedge \; i \;:\; 0 \leq i < N \;::\; m = \max(m, A[i]) \rangle$
   $\equiv\; \langle \wedge \; i \;:\; 0 \leq i < N \;::\; m \geq A[i] \rangle$
   $\equiv\; m \geq \langle \max \; i \;:\; 0 \leq i < N \;::\; A[i] \rangle$ $\bigtriangledown$

An important result dealing with fixed points is

Stability at fixed point
For any q,
$FP \;\wedge\; q$ is stable

5 A Program Structuring Mechanism: Union

As in other programming theories, it is often convenient to view or design a UNITY program as a composition of several program components. In this monograph, we consider a particularly simple kind of program composition: union. The union of programs F, G—denoted by $F \;[\!]\; G$—is obtained by appending the appropriate sections of their code. Union is defined only for those programs F, G whose *declare-* and *initial-section*s are not contradictory.) Hence the set of statements of $F \;[\!]\; G$ is the union of the statements in F nd G.

Programs F, G may be thought of as executing asynchronously in $F \parallel G$. The union operator is useful in understanding, among others, process networks in which each process may be viewed as a program and the entire network is their union. We see examples of union in Sections ? and ?. The following theorem is fundamental for understanding union (most parts of this theorem appear in Chapter 7 of []).

The Union Theorem:

•

$$\frac{\text{initially } p \text{ in } F}{\text{initially } p \text{ in } F \parallel G}$$

• p *unless* q in $F \parallel G$ $=$
  p *unless* q in F $\wedge$ p *unless* q in G
• p *ensures* q in $F \parallel G$ $=$
  [p *ensures* q in F $\wedge$ p *unless* q in G]
  $\vee$ [p *unless* q in F $\wedge$ p *ensures* q in G]
• *FP* of $F \parallel G$ $=$ *FP* of F $\wedge$ *FP* of G

Corollaries: From p. 156 of the book $\bigtriangledown$

Observe that if p names only local variables of F (i.e., variables that cannot be modified by any other program G), then p is stable in G. So we get

Corollary: from p.157 of the book $\bigtriangledown$

6 Termination Detection in a Ring

A message communicating system consists of a set of processes connected by direct channels. Each channel is directed from exactly one process to another; the messages in the channel are the ones setn by the former process to the latter which are, as yet, undelivered. A process is in one of two states: idel (i.e., quiescent) or nonidle. An idle process sends no message, and it remains idle as long as it receives no message. A nonidle process may become idle (autonomouisly). A message is received along a channel only if it had been sent along that channel. The system is *terminated* when all processes are idle and all channels are empty (i.e., have no messages in them) because all processes will stay idle—the first process to become nonidle has to first receive a message—and all channels will remain empty—idle processes send no message. It is required to develop an algorithm by which processes can determine that the system is terminated.

One of the more difficult aspects of this problem is to specify it precisely: What is given, what is to be designed and what should the design satisfy. We specify the problem in Section 5.1. We propose an algorithm for this problem in Section 5.2 and prove its correctness. The specification and verification rely heavily on the derived rules of UNITY logic.

In this monograph, we limit ourselves to a system in which the processes are connected in a unidirectional ring; the general case is described and solved in []. Our purpose here is primarily to acquaint the reader with UNITY-style specification, problem decomposition and proofs, and only secondarily with the problem of termination detection.

6.1 Problem Specification

Let there be N processes arranged in a unidirectional ring, the successor of the i^{th} process having index i'. The following variables are defined as follows—for all i (in this problem i is quantified over all indices in a given ring):

$s.i$, the number of messages sent by i (to i')
$r.i$, the number of messages received by i
$q.i$, a boolean variable that is true if and only if process i is idle

There is no notion of a process in UNITY. Hence we regard the entire process network as a program, D, that manipulates the variables given above, for all i. The following properties of D have been described earlier: the number of messages sent by a process is at least the number received by its successor, and both these numbers are nonnegative ($D1$, given below); the number of messages sent and received by a process are nondecreasing ($D2$); a process remains idle as long as it receives no message ($D3$); and an idle process has to become nonidle first in order to send a message ($D4$). Formally, for all i, we have the following properties in D.

$D1$. $s.i \geq r.i' \geq 0$ is invariant.
$D2$. $r.i \geq m$ is stable,
$s.i \geq n$ is stable
$D3$. $q.i \ \wedge \ r.i = m$ *unless* $r.i > m$
$D4$. $q.i \ \wedge \ s.i = n$ *unless* $\neg q.i \ \wedge \ s.i = n$

The properties, $D1 - D4$, are the only properties of D that we will assume. There are several aspects of the specification that are worth noting. First, we do not assume that the channels are FIFO, i.e., message sent along a channel may be delivered in a different order from the sending order. Second, there is no guarantee that a message will be delivered. Third, several processes may receive and/or send messages in one step; however a process may not receive a message, become nonidle and send a message all in one step (from $D4$ a process can send a message in a step provided it is nonidle at the beginning of the step).

Observation 1: Variables $s.i, r.i, q.i$, for all i, are local to D. Hence, applying locality corollary of the union theorem (Section 4), we derive that $D1 - D4$ are also properties of $D \,[\!]\, G$, for any G. $\bigtriangledown$

6.2 Definition of Termination

We have informally described termination as "all processes are idle and all channels are empty (for each channel, the number of messages sent equals the number received)." We show that none of the variables can change once the termination condition holds.

Let,

$$T \ \equiv \ \langle \forall \ i \ :: \ q.i \ \wedge \ s.i = r.i' \rangle$$

Observe that proving T to be stable only establishes that no $q.i$ will change (become false) once T holds; however, it is possible for $r.i, s.i$ to change while preserving $s.i = r.i'$, for all i. Hence, we prove below that $T \ \wedge \ \langle \forall \ i \ :: \ r.i = m.i \rangle$ is stable where $m.i$'s are arbitrary constants.

$D5$. $T \ \wedge \ \langle \forall \ i \ :: \ r.i = m.i \rangle$ is stable.

Proof of $D5$: The result is proven by taking the conjunction of $D3, D4$ and then taking the conjunction of the result over all i.

$q.i \ \wedge \ r.i = m.i \ \wedge \ s.i = m.i' \ unless \ r.i > m.i \ \wedge \ s.i = m.i'$
, replacing m, n by $m.i, m.i'$ in $D3, D4$ and then taking their conjunction
$\langle \forall \ i \ :: \ q.i \ \wedge \ r.i = m.i \ \wedge \ s.i = m.i' \rangle \ unless$
$\langle \forall \ i \ :: \ (q.i \ \wedge \ r.i = m.i \ \wedge \ s.i = m.i') \ \vee \ (r.i > m.i \ \wedge \ s.i = m.i') \rangle \ \wedge$
$\langle \exists \ i \ :: \ r.i > m.i \ \wedge \ s.i = m.i' \rangle$
, taking general conjunction of the above over all i (1)

The left side of (1)
$\equiv \quad \langle \forall \ i \ :: \ q.i \ \wedge \ s.i = r.i' \rangle \ \wedge \ \langle \forall \ i \ :: \ r.i = m.i \rangle$
$\equiv \quad T \ \wedge \ \langle \forall \ i \ :: \ r.i = m.i \rangle$

The first conjunct in the right side of (1) $\Rightarrow \ \langle \forall \ i \ :: \ s.i = m.i' \rangle$
The second conjunct in the right side of (1) $\Rightarrow \ \langle \exists \ i \ :: \ r.i > m.i \rangle$
Hence the right side of (1)
$\Rightarrow \quad \langle \forall \ i \ :: \ s.i = m.i' \rangle \ \wedge \ \langle \exists \ i \ :: \ r.i > m.i \rangle$
$\Rightarrow \quad \langle \exists \ i \ :: \ r.i' > m.i'' \ \wedge \ m.i' = s.i \rangle$
, replacing $\langle \exists \ i \ :: \ r.i > m.i \rangle$ by $\langle \exists \ i \ :: \ r.i' > m.i' \rangle$
$\equiv \quad false$
, using the substitution axiom and $s.i \geq r.i'$ from ($D1$) $\bigtriangledown$

$D6$. T is stable

Proof of $D6$: Eliminating the free variables $m.i$ (using the corresponding corollary from Sectoin ?) from $D5$. $\bigtriangledown$

6.3 An Algorithm for Termination Detection

Detection of T is nontrivial: An asynchronous inspection of the processes and channels may find every process to be idle and every channel to be empty; yet, T may not hold. An algorithm that does allow asynchronous inspection to succeed is the following.

Process i records the values of $q.i, r.i, s.i$ in variables $vq.i, vr.i, vs.i$, respectively at arbitrary times. Let VT be a predicate, analogous to T, defined as follows.

$$VT \ \equiv \ \langle \forall \ i \ :: \ vq.i \ \wedge \ vs.i = vr.i' \rangle$$

We will show that $VT \Rightarrow T$, i.e., detecting VT is sufficient to guarantee T. Detection of VT can be accomplished in a number of ways. A token may visit the processes collecting the values of $vq.i, vs.i, vr.i$, from each i, and whenever VT is satisfied by the values carried by the token, T can be asserted. Another possibility is to employ a central process to which all processes send their recorded values at arbitrary times, and the central process can compute VT. The reason VT is easier to compute than T is that the variables in VT are unaffected by execution program D whereas the variables in G are; direct computation of T would require us to suspend program D.

We view the recording programs of all processes together as a single program R. The entire program then is $D \parallel R$. Program R is given below. (Declarations of $vq.i, vs.i, vr.i$ are not shown; they are boolean, integer, integer, respectively.)

```
Program $R$
    initially  $\langle \parallel \ i \ :: \ vq.i, vs.i, vr.i \ := \ false, 0, 0 \rangle$
    assign     $\langle \parallel \ i \ :: \ vq.i, vs.i, vr.i \ := \ q.i, s.i, r.i \rangle$
end $\{R\}$
```

We show that VT holds only if T holds ($DR1$, given below) and VT holds within finite time of the holding of T ($DR2$).

$DR1$. $VT \;\Rightarrow\; T$ is invariant in $D \;[\!]\; R$
$DR2$. $T \;\mapsto\; VT$ in $D \;[\!]\; R$

It is important to note that the program $D \;[\!]\; R$ can be viewed in two different ways. It is the union of two tasks, D and R. Each of these tasks is executed by all processes, i.e., $D = \langle [\!]\; i \;::\; D_i \rangle$ where D_i is the given program for process i, and $R = \langle [\!]\; i \;::\; R_i \rangle$ where R_i is the component of the recording program executed by i. Therefore $D \;[\!]\; R$ may also be viewed as $\langle [\!]\; i \;::\; D_i \;[\!]\; R_i \rangle$ where $D_i \;[\!]\; R_i$ is the program (given and recording) for process i. The second way of structuring $D \;[\!]\; R$, around processes, is not convenient for proving $DR1$ and $DR2$. We will, however, decompose $D \;[\!]\; R$ in this manner when we implement the algorithm on a set of processes. The computational model allows us to choose the most convenient decomposition for the purpose at hand.

6.4 Proof of the Algorithm

We note the two properties of $D \;[\!]\; R$. Property $DR3$ is easily understood; $DR4$ says that any process that was idle at the time of its last recording and that has not received any message since then is still idle and has not sent any message since then.

$DR3$. $s.i \geq vs.i \geq 0$ is invariant in $D \;[\!]\; R$,
$r.i \geq vr.i \geq 0$ is invariant in $D \;[\!]\; R$
$DR4$. $vq.i \;\wedge\; (r.i = vr.i) \;\Rightarrow\; q.i \;\wedge\; (s.i = vs.i)$ is invariant in $D \;[\!]\; R$

These properties can be proven by showing that they are invariant in R, from the text of R, and stable in D, and then applying corollary of the union theorem. Most of these proofs are straightforward; we show that the predicate in $DR4$ is stable in program D.

6.4.1 A Stability Proof in D

Lemma: $vq.i \;\wedge\; (vr.i = r.i) \;\Rightarrow\; q.i \;\wedge\; (vs.i = s.i)$ is stable in D
Proof: All properties in the following proof are of D.

$q.i \;\wedge\; r.i = m \;\wedge\; s.i = n$ *unless* $r.i > m$
, conjunction of $D3, D4$ and then weakening the consequence
$r.i > m$ is stable
, $D2$ with $m+1$ substituted for m
$(r.i > m) \;\vee\; (q.i \wedge r.i = m \wedge s.i = n)$ is stable
, disjunction of the above two
$\neg vq.i$ is stable
, $vq.i$ is constant in D
$vq.i \;\wedge\; r.i \leq m \;\Rightarrow\; q.i \;\wedge\; r.i = m \;\wedge\; s.i = n$ is stable
, simple disjunction of the above two and rewriting
$vq.i \;\wedge\; r.i \leq vr.i \;\Rightarrow\; q.i \;\wedge\; r.i = vr.i \;\wedge\; s.i = vw.i$ is stable
, substituting $vr.i, vs.i$ that are constants in D for m, n, respectively
$vq.i \;\wedge\; r.i = vr.i \;\Rightarrow\; q.i \;\wedge\; r.i = vr.i \;\wedge\; s.i = vs.i$ is stable
, substitution axiom on $DR3$ yields $r.i \leq vr.i \;\equiv\; r.i = vr.i$
$vq.i \;\wedge\; r.i = vr.i \;\Rightarrow\; q.i \;\wedge\; s.i = vs.i$ is stable
, simplifying the last predicate ∇

6.4.2 Proof of $DR1$

To prove that

$DR1$. $VT \;\Rightarrow\; T$ is invariant in $D \parallel R$

We show that

> $VT \;\Rightarrow\; T$ is stable in D and,
> $VT \;\Rightarrow\; T$ is invariant in R

Then, applying corollary of the union theorem, $DR1$ follows.

Proof of $VT \;\Rightarrow\; T$ is stable in D

$\neg VT$ is stable in D	, no variable of VT is modified
T is stable in D	, from ($D6$)
$VT \;\Rightarrow\; T$ is stable in D	, simple disjunction of the above two and rewriting $\quad\triangledown$

Proof of $VT \;\Rightarrow\; T$ is invariant in R

In the following proof, all properties are in R.
Initially $\neg VT$ holds because all $\neg vq.i$, for all i.
To show the stability of $VT \;\Rightarrow\; T$, we have to show for any arbitrary j that
$\{VT \Rightarrow T\}\;\; vq.j, vs.j, vr.j \;:=\; q.j, s.j, r.j \;\; \{VT \Rightarrow T\}$
We show the stronger assertion
$\{true\}\;\; vq.j, vs.j, vr.j \;:=\; q.j, s.j, r.j \;\; \{VT \Rightarrow T\}$
or equivalently, since the assignment establishes the postcondition
$\{(vq.j, vs.j, vr.j) =$
$(q.j, s.j, r.j)\}$
$VT \;\wedge\; (vq.j, vs.j, vr.j) = (q.j, s.j, r.j) \;\Rightarrow\; T$

The proof is as follows. Assume the antecedent of the above.

$vr.j' = vs.j$	, from VT in the antecedent
$vs.j = s.j$	, from the antecedent
$s.j \geq r.j'$	, from $D1$
$r.j' \geq vr.j'$	, from $DR3$
$vr.j' = r.j'$	, from the above four properties
$vq.j'$	, from VT
$q.j' \;\wedge\; s.j' = vs.j'$	, from $vq.j' \;\wedge\; r.j' = vr.j'$ using $DR4$
$(vq.j', vs.j', vr.j') = (q.j', s.j', r.j')$	, from the above three

Hence we have

$$VT \;\wedge\; (vq.j, vs.j, vr.j) = (q.j, s.j, r.j) \;\Rightarrow\; VT \wedge (vq.j', vs.j', vr.j') = (q.j', s.j', r.j')$$

Applying induction,

$$VT \;\wedge\; (vq.j, vs.j, vr.j) = (q.j, s.j, r.j) \;\Rightarrow\; VT \wedge (vq.i, vs.i, vr.i) = (q.i, s.i, r.i)\rangle$$

The consequent of the above implies T. $\quad\triangledown$

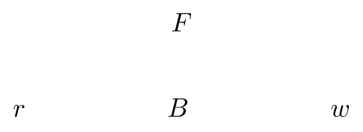

Figure 2: A Buffer Program B and its environment F

6.4.3 Proof of DR2

Let $u.i \equiv (vq.i, vs.i, vr.i) = (q.i, s.i, r.i)$

We show,

$T \mapsto T \wedge u.i$ in $D \parallel R$	(1)
$T \wedge u.i$ is stable in $D \parallel R$	(2)
$T \mapsto \langle \forall i :: T \wedge u.i \rangle$ in $D \parallel R$	, applying the completion rule for *leads-to* (Section ?)
$\langle \forall i :: T \wedge u.i \rangle \Rightarrow VT$	, from definitions of $T, u.i, VT$
$T \mapsto VT$ in $D \parallel R$	, from the above two

Proof of (1): $T \mapsto T \wedge u.i$ in $D \parallel R$

T *ensures* $T \wedge u.i$ in R	, from the text of R
T is stable in D	, from $D5$
T *ensures* $T \wedge u.i$ in $D \parallel R$	, corollary to the union theorem
$T \mapsto T \wedge u.i$ in $D \parallel R$	, definition of $\mapsto$ $\bigtriangledown$

Proof of (2): $T \wedge u.i$ is stable in $D \parallel R$

$T \wedge r.i = m.i \wedge r.i' = m.i'$ is stable in D
  , eliminating all other $m.j$ from $D5$
$vq.i$ is stable in D
  , $vq.i$ is constant in D
$T \wedge (q.i, s.i, r.i) = (vq.i, m.i', m.i)$ is stable in D
  , conjunction of the above two and rewriting
$T \wedge u.i$ is stable in D
  , replacing $m.i, m.i'$ by $vr.i$ and $vs.i$ that are constants in D
$T \wedge u.i$ is stable in R
  , from the text of R
$T \wedge u.i$ is stable in $D \parallel R$
  , corollary of the union theorem $\bigtriangledown$

7 Notes on UNITY 02-88

8 The Buffer Program

A *buffer* program, B, reads data from a variable r and writes data into a variable w. The program runs asynchronously with another program, called F, which writes into r and reads from w. The communication structure is shown in Fig. 1.

The program F represents both the producer and the consumer. There might be multiple producers and consumers or even a single process that is both the producer and the consumer; the exact number is irrelevant for the specification of the buffer. The values that can be written into r, w are, again, irrelevant for specification. However we do postulate a special value, $\emptyset$, which is written into a variable to denote that it is "empty," i.e., it contains no useful data. The protocol for reading and writing is as follows. Program F writes into r only if $r = \emptyset$; only if $r \neq \emptyset$, program B reads a value from r and it may set r to $\emptyset$. Program B stores a value in w only if w is $\emptyset$; program F reads from w and it may set w to $\emptyset$ to indicate that it is ready to consume the next piece of data.

8.1 Auxiliary Variables

It is convenient to employ the auxiliary variables, $\bar{r}$ and $\bar{w}$, which denote, respectively, the sequence of data items (i.e., non-$\emptyset$ values) written into r and w. The typical definitions of $\bar{r}, \bar{w}$ are given by augmenting the program text; a value d, $d \neq \emptyset$, is appended to $\bar{r}$ whenever d is stored in r. Fortunately, our logical operators provide a preferable way to define auxiliary variables.

Notation: All through this paper d, e, f refer to arbitrary data values or $\emptyset$, and x, y denote sequences of these values. The concatenation operator for sequences is ";" . For all x, $(x;\emptyset) = (\emptyset;x) = x$. $\bigtriangledown$

Definitions of $\bar{r}, \bar{w}$ are as follows. (The two definitions are completely similar.)

$$\text{initially} \quad (\bar{r}, \bar{w}) = (r, w) \text{ in } B \parallel F \tag{A0}$$
$$\bar{r} = x \ \wedge \ r = d \ \ \textit{unless} \ \ \bar{r} = x; r \ \wedge \ r \neq d \text{ in } B \parallel F \tag{A1}$$
$$\bar{w} = y \ \wedge \ w = e \ \textit{unless} \ \ \bar{w} = y; w \ \wedge \ w \neq e \text{ in } B \parallel F \tag{A2}$$

To understand A1, note that if $\emptyset$ is stored in r (i.e., data is consumed from r) then $\bar{r}$ remains unchanged (because $x; r = x; \emptyset = x$). If r is changed to d, $d \neq \emptyset$, then $\bar{r}$ is extended by d. These properties, A0,A1,A2, may be taken as axioms in a proof of $B \parallel F$.

8.2 Specification of a Buffer of Size N

A buffer of size N, $N > 1$, has $N-1$ internal words for storage. (The reason for using $N-1$, rather than N, is that with this definition concatenations of buffers of size M, N results in a buffer of size $M + N$.) For $N = 1$, the buffer program simply moves data from r to w.

Notation: Define an ordering relation, $\prec$, among data values as follows:

$\emptyset$ is "smaller than" all non-$\emptyset$ values, i.e.,
$$d \ \prec \ e \quad \equiv \quad d = \emptyset \ \wedge \ e \neq \emptyset$$ $\bigtriangledown$

The properties P1 and P2, given below, state respectively that program B removes only non-$\emptyset$ data from r and it may set r to $\emptyset$, and it writes only non-$\emptyset$ data values in w provided $w = \emptyset$.

$$\text{initially } w = \emptyset \text{ in } B \tag{P0}$$
$$r = d \ \ \textit{unless} \ \ r \prec d \text{ in } B \tag{P1}$$
$$w = e \ \ \textit{unless} \ \ e \prec w \text{ in } B \tag{P2}$$

Observe that setting d to $\emptyset$ in (P1) gives us (because $r \prec \emptyset \equiv \textit{false}$),
$r = \emptyset$ is stable in B

That is, B never changes r from $\emptyset$ to non-$\emptyset$. Similarly, we may deduce from (P2) that B never overwrites a non-$\emptyset$ value in w.

The next property, P3, says that (1) the sequence of data items stored in w by B is a prefix of the sequence supplied to it in r, (2) these two sequences do not differ by more than N in length, (3) if the two sequences differ by less than N in length (i.e., internal buffer is nonfull) or w is $\emptyset$ then data, if any, would be removed from r, (4) if more items have been supplied to B than have been produced or r is non-$\emptyset$ then w is or will be set to a non-$\emptyset$ value. These properties, however, cannot hold if program F, with which B is composed, is uncooperative; for instance, if F overwrites a data value in r with another data value then B can never reproduce the overwritten value in w. Hence these properties—collectively called the conclusion for B, or $B.conc$—hold conditioned upon two properties of F—collectively called the hypothesis for B, or $B.hypo$—that F writes a value into r only if r is $\emptyset$ and F sets w only to $\emptyset$.

Notation: $|\bar{r}|, |\bar{w}|$ denote the lengths of $\bar{r}, \bar{w}$, respectively. $\bigtriangledown$

The property (P3) is,

$$\langle \forall\ F\ ::\ B.hypo \text{ in } F \quad \Rightarrow \quad B.conc \text{ in } B\ [\!]\ F \rangle \tag{P3}$$

where,

$$\begin{array}{lll} B.hypo & :: & r = d \ \ \textit{unless}\ \ d \prec r, \\ & & w = e \ \ \textit{unless}\ \ w \prec e \end{array}$$

and,

$$\begin{array}{lllll} B.conc & :: & \bar{w} \subseteq \bar{r}, & & \\ & & |\bar{r}| \le |\bar{w}| + N, & & \\ & & |\bar{r}| < |\bar{w}| + N & \vee & w = \emptyset \ \mapsto\ r = \emptyset, \\ & & |\bar{r}| > |\bar{w}| & \vee & r \ne \emptyset \ \mapsto\ w \ne \emptyset \end{array}$$

Note that P3 is a property of program B. It says that if B is composed with any program F that satisfies $B.hypo$, then $B\ [\!]\ F$ satisfies $B.conc$. Also observe that $B.hypo$ is symmetric to (P1,P2), because the protocols for production and consumption by F are symmetric to those of B, with the roles of r, w interchanged.

This specification makes no commitment about the internal structure of the buffer program. For instance, the buffer program may move a data value to the adjacent buffer space provided the latter is $\emptyset$; or, it may move a data value as far as possible toward the destination. Similarly, little assumption is made about the internal structure of F.

9 Buffer Concatenation

Let $B1$ be a buffer of size M with input and output words, r, s, respectively, and $B2$ be a buffer of size N with input and output words s, w, respectively. We show that $B1\ [\!]\ B2$ implements a buffer of size $M + N$ with input and output words r, w, respectively. The arrangement is shown pictorially in Fig. 2.

9.1 Proof of P0 for B

Trivially, from P0 of $B2$.

9.2 Proof of P1,P2 for B

We are given

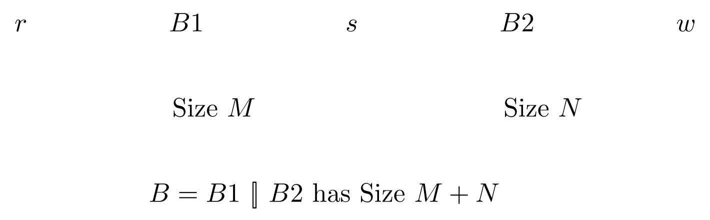

Figure 3: Concatenations of buffers $B1, B2$

$$\begin{array}{ll}
r = d \ \ \textit{unless} \ \ r \prec d \text{ in } B1 & \text{(P1 for } B1\text{)} \\
s = f \ \ \textit{unless} \ \ f \prec s \text{ in } B1 & \text{(P2 for } B1\text{)}
\end{array}$$

and

$$\begin{array}{ll}
s = f \ \ \textit{unless} \ \ s \prec f \text{ in } B2 & \text{(P1 for } B2\text{)} \\
w = e \ \ \textit{unless} \ \ e \prec w \text{ in } B2 & \text{(P2 for } B2\text{)}
\end{array}$$

We have to show, P1,P2 for B, where $B = B1 \parallel B2$:

$$\begin{array}{ll}
r = d \ \ \textit{unless} \ \ r \prec d \text{ in } B1 \parallel B2 & \text{(P1 for } B\text{)} \\
w = e \ \ \textit{unless} \ \ e \prec w \text{ in } B1 \parallel B2 & \text{(P2 for } B\text{)}
\end{array}$$

We only show the proof of P1; P2's proof is nearly identical.

$$\begin{array}{ll}
r = d \ \ \textit{unless} \ \ r \prec d \text{ in } B1 & \text{, from P1 of } B1 \\
r = d \text{ is stable in } B2 & \text{, } r \text{ is not accessible to } B2 \\
r = d \ \ \textit{unless} \ \ r \prec d \text{ in } B1 \parallel B2 & \text{, from Corollary 1 to union theorem}
\end{array}$$

9.3 Proof of P3 for B

We are required to prove P3 for B assuming P3 for $B1$ and $B2$ (and also using P1,P2 for both $B1, B2$). More precisely, we have to show:

$$\begin{array}{rl}
[& \langle \forall\, G \ :: \ B1.\textit{hypo} \text{ in } G \ \Rightarrow \ B1.\textit{conc} \text{ in } B1 \parallel G \rangle \\
 & \wedge \langle \forall\, H \ :: \ B2.\textit{hypo} \text{ in } H \ \Rightarrow \ B2.\textit{conc} \text{ in } B2 \parallel H \rangle] \\
\Rightarrow & \langle \forall\, F \ :: \ B.\textit{hypo} \text{ in } F \ \ \Rightarrow \ B.\textit{conc} \text{ in } B \parallel F \rangle
\end{array}$$

Equivalently,

$$\begin{array}{l}
\langle \forall\, F \ :: \ [B.\textit{hypo} \text{ in } F \ \ \wedge \ \langle \forall\, G \ :: \ B1.\textit{hypo} \text{ in } G \Rightarrow \ B1.\textit{conc} \text{ in } B1 \parallel G \rangle \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge \ \langle \forall\, H \ :: \ B2.\textit{hypo} \text{ in } H \Rightarrow \ B2.\textit{conc} \text{ in } B2 \parallel H \rangle] \\
\qquad \Rightarrow \ B.\textit{conc} \text{ in } B \parallel F \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (1) \\
\rangle
\end{array}$$

The properties $B1.\textit{hypo}, B1.\textit{conc}$ (and similarly $B2.\textit{hypo}, B2.\textit{conc}$) are obtained from $B.\textit{hypo}, B.\textit{conc}$ of Section 3.2 by replacing w by s; these are shown below.

$$\begin{array}{ll}
B1.\textit{hypo} \ :: & r = d \ \ \textit{unless} \ \ d \prec r, \\
 & s = f \ \ \textit{unless} \ \ s \prec f, \\
B1.\textit{conc} \ :: & \bar{s} \subseteq \bar{r}, \\
 & |\bar{r}| \leq |\bar{s}| + M, \\
 & |\bar{r}| < |\bar{s}| + M \ \vee \ s = \emptyset \ \mapsto \ r = \emptyset, \\
 & |\bar{r}| > |\bar{s}| \qquad\ \ \vee \ r \neq \emptyset \ \mapsto \ s \neq \emptyset
\end{array}$$

$$\begin{array}{lll} B2.\mathit{hypo} & :: & s = f \quad \mathit{unless} \quad f \prec s, \\ & & w = e \quad \mathit{unless} \quad w \prec e \\ B2.\mathit{conc} & :: & \bar{w} \subseteq \bar{s}, \\ & & |\bar{s}| \leq |\bar{w}| + N, \\ & & |\bar{s}| < |\bar{w}| + N \vee w = \emptyset \;\mapsto\; s = \emptyset, \\ & & |\bar{s}| > |\bar{w}| \qquad\quad \vee\, s \neq \emptyset \;\mapsto\; w \neq \emptyset \end{array}$$

The structure of the proof is as follows. Consider an arbitrary F and let $G = B2 \parallel F$ and $H = B1 \parallel F$ (because $B2 \parallel F$ is the environment for $B1$ and $B1 \parallel F$ for $B2$). We show,

$$B.\mathit{hypo} \text{ in } F \Rightarrow B1.\mathit{hypo} \text{ in } G \quad \text{and,} \tag{2}$$
$$B.\mathit{hypo} \text{ in } F \Rightarrow B2.\mathit{hypo} \text{ in } H \tag{3}$$

Hence, from the antecedent of (1), we may then assume $B1.\mathit{conc}$ in $B1 \parallel G$ and $B2.\mathit{conc}$ in $B2 \parallel H$. ¿From the definitions of B, G, H, we have $B1 \parallel G = B2 \parallel H = B1 \parallel B2 \parallel F = B \parallel F$. Hence, we have $B1.\mathit{conc}$ in $B \parallel F$ and $B2.\mathit{conc}$ in $B \parallel F$. We will next deduce,

$$B1.\mathit{conc} \text{ in } B \parallel F \;\wedge\; B2.\mathit{conc} \text{ in } B \parallel F \;\Rightarrow\; B.\mathit{conc} \text{ in } B \parallel F \tag{4}$$

Proof of (2)

The two proofs, corresponding to the two conjuncts in $B1.\mathit{hypo}$, are shown below.

Proof of $r = d$ *unless* $d \prec r$ in G

$r = d \quad \mathit{unless} \quad d \prec r$ in F	, from $B.\mathit{hypo}$ in F
$r = d$ is stable in $B2$	, r is not accessed in $B2$
$r = d \quad \mathit{unless} \quad d \prec r$ in $B2 \parallel F$	, corollary to union theorem $\quad\bigtriangledown$

Proof of $s = f$ *unless* $s \prec f$ in G

$s = f \quad \mathit{unless} \quad s \prec f$ in $B2$	, property P2 of $B2$
$s = f$ is stable in F	, s is not accessed in F
$s = f \quad \mathit{unless} \quad s \prec f$ in $B2 \parallel F$	, corollary to union theorem $\quad\bigtriangledown$

Proof of (3) is similar to proof of (2).

Proof of (4)

The proof has four parts, corresponding to the four conjuncts in $B.\mathit{conc}$. All properties are of $B \parallel F$.

Proof of $\bar{w} \subseteq \bar{r}$

$\bar{w} \subseteq \bar{s}$	, from $B2.\mathit{conc}$
$\bar{s} \subseteq \bar{r}$	, from $B1.\mathit{conc}$
$\bar{w} \subseteq \bar{r}$	, from the above two $\quad\bigtriangledown$

Proof of $|\bar{r}| \leq |\bar{w}| + M + N$

$|\bar{r}| \leq |\bar{s}| + M$	, from $B1.\mathit{conc}$
$|\bar{s}| \leq |\bar{w}| + N$	, from $B2.\mathit{conc}$
$|\bar{r}| \leq |\bar{w}| + M + N$	, from the above two $\quad\bigtriangledown$

Proof of $|\bar{r}| < |\bar{w}| + M + N \;\vee\; w = \emptyset \;\mapsto\; r = \emptyset$

$|\bar{s}| < |\bar{w}| + N \;\vee\; w = \emptyset \mapsto \; s = \emptyset$
, from $B2.conc$
$s = \emptyset \qquad\qquad\qquad \mapsto \; r = \emptyset$
, from $B1.conc$ using implication rule for *leads-to*
$|\bar{s}| < |\bar{w}| + N \;\vee\; w = \emptyset \mapsto \; r = \emptyset$
, transitivity on the above two
$|\bar{r}| < |\bar{s}| + M \;\mapsto\; r = \emptyset$
, from $B1.conc$ using implication rule for *leads-to*
$|\bar{r}| < |\bar{s}| + M \;\vee\; |\bar{s}| < |\bar{w}| + N \;\vee\; w = \emptyset \;\mapsto\; r = \emptyset$
, disjunction on the above two
$|\bar{r}| < |\bar{w}| + M + N \quad\Rightarrow\quad |\bar{r}| < |\bar{s}| + M \;\vee\; |\bar{s}| < |\bar{w}| + N$
, seen easily by taking the contrapositive
$|\bar{r}| < |\bar{w}| + M + N \;\vee\; w = \emptyset \;\mapsto\; r = \emptyset$
, from the above two using implication rule for *leads-to* ▽

Proof of $|\bar{r}| < |\bar{w}| \;\vee\; r \neq \emptyset \;\mapsto\; w \neq \emptyset$
Similar to the above proof. ▽

10 A Refinement of the Specification

As a first step toward implementing a buffer, we propose a more refined (i.e., stronger) specification of a buffer. Our proof obligation then is to show that this proposed specification implies the specification given by properties P0,P1,P2,P3, of Section 3.2. In the next section we show a program that implements this refined specification. Because of the result of Section 4, any buffer of size N can be implemented by concatenating N buffers of size 1; hence we limit our refinement to $N = 1$.

For $N = 1$, the buffer program has no internal words for data storage (recall that there are $N - 1$ internal words for storage). Hence the only strategy for the buffer program is to move data from r to w when $w \neq \emptyset$. This is captured in the following specification.

initially $w = \emptyset$ in B (R0)
$(r, w) = (d, e)$ *unless* $r \prec d \;\wedge\; e \prec w \;\wedge\; w = d$ in B (R1)
$w \prec r$ *ensures* $r \prec w$ in B (R2)

Property R1 says that the pair (r, w) changes only if a non-$\emptyset$ data value is moved from r to w when the latter is $\emptyset$. The progress property, R2, says that if data can be moved—i.e., $r \neq \emptyset \;\wedge\; w = \emptyset$—then it will be moved.

Now we show that the refinement is correct, i.e. (R0,R1,R2) imply (P0,P1,P2,P3), given in Section 3.2.

10.1 Proofs of P0,P1,P2

Proof of (P0): initially $w = \emptyset$ in B

Immediate from R0. ▽

Proof of (P1): $r = d$ *unless* $r \prec d$ in B

$r = d \;\wedge\; w = e$ *unless* $r \prec d$ in B
, from R1 by weakening its consequence
$r = d$ *unless* $r \prec d$ in B
, eliminating free variable e ▽

Proof of (P2): $w = e$ *unless* $e \prec w$ in B

$r = d \ \wedge \ w = e$ *unless* $e \prec w$ in B

  , from R1 by weakening its consequence

$w = e$ *unless* $e \prec w$ in B

  , eliminating free variable d $\bigtriangledown$

10.2 Proof of P3

The property P3 is of the form,

$$\langle \forall\ F\ ::\ B.hypo \text{ in } F\ \Rightarrow\ B.conc \text{ in } B \parallel F \rangle$$

We prove P3 from (R0,R1,R2) by first proving that

$$\langle \forall\ F\ ::\ B.hypo \text{ in } F \wedge (\text{R0,R1}) \Rightarrow\ (\bar{r} = \bar{w}; r \text{ in } B \parallel F)\rangle \qquad \text{(P4)}$$

Then we show that

$$B.hypo \text{ in } F \wedge (\text{R2}) \wedge (\bar{r} = \bar{w}; r \text{ in } B \parallel F)\ \Rightarrow\ (B.conc \text{ in } B \parallel F) \qquad \text{(P5)}$$

10.2.1 Proof of P4

To show that $\bar{r} = \bar{w}; r$ is invariant in $B \parallel F$, the proof obligations are,

  initially $\bar{r} = \bar{w}; r$ in $B \parallel F$ and,
  $\bar{r} = \bar{w}; r$ is stable in $B \parallel F$

The initial condition can be seen from,

  initially $(\bar{r}, \bar{w}) = (r, w)$ in $B \parallel F$, from (A0) of Section 3.1
  and, initially $w = \emptyset$ in $B \parallel F$, from (R0) and the rule for union of B, F

Next we show that $\bar{r} = \bar{w}; r$ is stable in $B \parallel F$.

Proof of $\bar{r} = \bar{w}; r$ is stable in $B \parallel F$

$r = d$ *unless* $d \prec r$ in F

  , from $B.hypo$ in F

$w = e$ *unless* $w \prec e$ in F

  , from $B.hypo$ in F

$(r, w) = (d, e)$ *unless* $(d, w) \prec (r, e)$ in F

  , conjunction of the above two; $(d, w) \prec (r, e)$ stands for
   $(d = r \ \wedge \ w \prec e) \ \vee \ (d \prec r \ \wedge \ w = e) \ \vee \ (d \prec r \ \wedge \ w \prec e)$

$(r, w) = (d, e)$ *unless* $r \prec d \ \wedge \ e \prec w \ \wedge \ w = d$ in B

  , from (R1)

$(r, w) = (d, e)$ *unless*
  $(d, w) \prec (r, e)$ {1.1}
  $\vee\ (r \prec d \ \wedge \ e \prec w \ \wedge \ w = d)$ {1.2}
   in $B \parallel F$ (1)

  , weakening the right sides of the above two and applying the union theorem

Next use the properties (A1,A2) of Section 3.1 and form their conjunction to get:

$(\bar{r}, \bar{w}) = (x, y) \ \wedge \ (r, w) = (d, e)$ *unless*
  $[(\bar{r}, \bar{w}) = (x, y; w) \ \wedge \ r = d \ \wedge \ w \neq e]$ {2.1}
  $\vee\ [(\bar{r}, \bar{w}) = (x; r, y) \ \wedge \ r \neq d \ \wedge \ w = e]$ {2.2}
  $\vee\ [(\bar{r}, \bar{w}) = (x; r, y; w) \ \wedge \ r \neq d \ \wedge \ w \neq e]$ {2.3}
   in $B \parallel F$ (2)

Next form the conjunction of (1) and (2). Observe that every disjunct in the right side of (1) and (2) imply $(r, w) \neq (d, e)$, and therefore, conjunctions of these with the left sides of (2) and (1), respectively, result in *false*.

$$
\begin{array}{lr}
(\bar{r}, \bar{w}) = (x, y) \ \wedge \ (r, w) = (d, e) \ \textit{unless} & \\
\quad [(\bar{r}, \bar{w}) = (x, y; w) \ \wedge \ r = d \ \wedge \ w \prec e] & \{3.1, \text{from } 1.1 \wedge 2.1\} \\
\vee \ [(\bar{r}, \bar{w}) = (x; r, y) \ \wedge \ d \prec r \ \wedge \ w = e] & \{3.2, \text{from } 1.1 \wedge 2.2\} \\
\vee \ [(\bar{r}, \bar{w}) = (x; r, y; w) \ \wedge \ d \prec r \ \wedge \ w \prec e] & \{3.3, \text{from } 1.1 \wedge 2.3\} \\
\vee \ [(\bar{r}, \bar{w}) = (x; r, y; w) \ \wedge \ r \prec d \ \wedge \ e \prec w \ \wedge \ w = d] & \{3.4, \text{from } 1.2 \wedge 2.3\} \\
\qquad \text{in } B \parallel F & (3)
\end{array}
$$

We next set the free variable x to $y; d$ in (3). The terms in the right side may be weakened to yield

$\bar{r} = \bar{w}; r \ \wedge \ (\bar{w}, r, w) = (y, d, e)$ *unless* $\bar{r} = \bar{w}; r \ \wedge \ (\bar{w}, r, w) \neq (y, d, e)$ in $B \parallel F$
$\bar{r} = \bar{w}; r$ is stable in $B \parallel F$, Corollary 2 of Section 2.1.5 $\bigtriangledown$

10.2.2 Proof of P5

Proof of $B.hypo$ in $F \wedge$ (R2) $\wedge$ ($\bar{r} = \bar{w}; r$ in $B \parallel F$) $\Rightarrow$ ($B.conc$ in $B \parallel F$)

The proof consists of four parts; each part establishes one of the conjuncts in $B.conc$.

Proof of $\bar{w} \subseteq \bar{r}$ in $B \parallel F$

Immediate from $\bar{r} = \bar{w}; r$ in $B \parallel F$ $\bigtriangledown$

Proof of $|\bar{r}| \leq |\bar{w}| + 1$ in $B \parallel F$

$|\bar{r}| = |\bar{w}| + |r|$ in $B \parallel F$, from $\bar{r} = \bar{w}; r$ in $B \parallel F$
$|\bar{r}| \leq |\bar{w}| + 1$ in $B \parallel F$, $|r| \leq 1$ $\bigtriangledown$

The remaining two progress properties in $B.conc$ can be established by first proving
$w \prec r \ \mapsto \ r \prec w$ in $B \parallel F$ (P6)

Proof of (P6): $w \prec r \ \mapsto \ r \prec w$ in $B \parallel F$
$r = d$ *unless* $d \prec r$ in F
, from $B.hypo$ in F
$d \neq \emptyset$ is stable in F
, d is constant in F
$r \neq \emptyset \ \wedge \ r = d$ is stable in F
, conjunction of the above two
$r \neq \emptyset$ is stable in F
, eliminating free variable d (1)
$w = e$ *unless* $w \prec e$ in F
, from $B.hypo$ in F
$w = \emptyset$ is stable in F
, setting e to $\emptyset$ in the above
$w \prec r$ is stable in F
, conjunction of (1) and the above
$w \prec r$ *ensures* $r \prec w$ in B
, from (R2)
$w \prec r$ *ensures* $r \prec w$ in $B \parallel F$
, Corollary 2 of the union theorem on the above two
$w \prec r \ \mapsto \ r \prec w$ in $B \parallel F$
, from the definition of $\mapsto$ $\bigtriangledown$

In UNITY, a "substitution axiom" allows us to replace a predicate p by another predicate q (and vice-versa) anywhere in the proof of a given program provided $p \equiv q$ is an invariant of the program. Since $\bar{r} = \bar{w}; r$ is an invariant of $B \; [\!] \; F$, we have

$$r = \emptyset \quad \equiv \quad |\bar{r}| < |\bar{w}| + 1 \text{ in } B \; [\!] \; F \tag{P7}$$

We use the substitution axiom in the following proof to replace $r = \emptyset$ by $|\bar{r}| < |\bar{w}| + 1$. We show one progress proof only; the other one is similar. In the following proof, all properties are of $B \; [\!] \; F$.

Proof of $|\bar{r}| < |\bar{w}| + 1 \; \vee \; w = \emptyset \; \mapsto \; r = \emptyset$

$r \neq \emptyset \; \wedge \; w = \emptyset \mapsto \; r = \emptyset$, expanding (P6) and using the implication rule
$r = \emptyset \qquad \mapsto \; r = \emptyset$, implication rule
$r = \emptyset \; \vee \; w = \emptyset \mapsto \; r = \emptyset$, disjunction of the above two
$|\bar{r}| < |\bar{w}| + 1 \; \vee \; w = \emptyset \; \mapsto \; r = \emptyset$, using (P7) to replace $r = \emptyset$ by $|\bar{r}| < |\bar{w}| + 1$ $\bigtriangledown$

11 An Implementation

The specification of Section 5 can be implemented by a program whose only statement moves data from r to w provided $w = \emptyset$ (if $r = \emptyset$, the movement has no effect):

$$r, w \; := \; \emptyset, r \qquad \text{if} \quad w = \emptyset$$

The proof that this fragment has the properties (R1,R2) is immediate from the definition of *unless* and *ensures*. The initial condition of this program is $w = \emptyset$, and hence (R0) is established.

The implementation for buffer of size N, $N > 1$, is the union of N such statements: one statement each for moving data from a location to an adjacent location (closer to w) provided the latter is $\emptyset$. We show how this program may be expressed in the UNITY programming notation.

Rename the variables r and w to be $b[0]$ and $b[N]$, respectively. The internal buffer words are $b[1]$ through $b[N-1]$. In the following program we write $\langle [\!] \; i \; : \; 0 < i \leq N \; :: \; t(i) \rangle$ as a shorthand for $t(1) \; [\!] \; t(2) \ldots [\!] \; t(N)$, where $t(1)$, for instance, is obtained by replacing every occurrence of i by 1 in $t(i)$. The program specifies the initial values of $b[1]$ through $b[N]$ to be $\emptyset$ (in the part followed by **initially**). The statements of the program are given after **assign**; the generic statement shown moves $b[i-1]$ to $b[i]$ provided the latter is $\emptyset$.

Program buffer {of N words, $N \geq 1$}

initially $\langle [\!] \; i \; : \; 0 < i \leq N \; :: \; b[i] = \emptyset \rangle$

assign $\langle [\!] \; i \; : \; 0 < i \leq N \; :: \; b[i-1], b[i] \; := \; \emptyset, b[i-1] \qquad \text{if} \quad b[i] = \emptyset \rangle$

end {**buffer**}