Multiprocessor Implementation of Nondeterminate Computations in a Functional Programming Framework

Jack B. Dennis
MIT Laboratory for Computer Science
Cambridge, MA, 02139, USA
dennis@aj.lcs.mit.edu
and
Guang R. Gao
School of Computer Science, McGill University
Montreal, Canada
gao@andy.cs.mcgill.ca

Abstract

Nondeterminacy arises when the outcome of a computation may be different for different runs with the same input. Generally, nondeterminacy is undesirable because its presence makes correctness of computations difficult to verify, and bugs difficult to track down. Yet nondeterminate computation is useful in some programs such as those for transaction processing.

Functional programming is uniquely capable of achieving highly parallel program execution without allowing the possibility of nondeterminate results. In this paper, we discuss how functional programming models may be augmented to support nondeterminate computation. We begin with a brief review of past concepts and proposals for expressing nondeterminate computation. Given the hypothesis that functional programming is the best programming model for expressing determinate computation, we introduce a basic functional programming model and its implementation by means of an abstract computer system. We show how this abstract computation model may be augmented to incorporate support for nondeterminate computations by the simple addition of a *swap* memory operation. The remainder of the paper illustrates application of this base to a transaction processing example, and compares this approach with solutions using M-structures, as incorporated into the Ph language, and with solutions for a conventional shared-memory programming environment.

1 Introduction

Nondeterminacy arises when the outcome of a computation may be different for different runs with the same input. A good example is transaction processing, where the order in which commands take effect on a database depends on the timing of asynchronous events within the computer system. Generally, nondeterminacy is undesirable because its presence makes correctness of computations difficult to verify, and bugs difficult to track down. Yet nondeterminate computation is useful in some programs such as those for transaction processing. Furthermore, in some kinds of computing, such as heuristic search problems, use of nondeterminacy can lead to more efficient problem solving with parallel computers.

We believe computer systems should be designed so that programmers can implement computations that do not require nondeterminacy in a language or style that precludes nondeterminate effects. The popular practical programming languages do not offer such a guarantee when extended for the writing of parallel programs. This is because explicit process synchronization commands are used which introduce the possibility of nondeterminate behavior, even when it may not be desired.[1] Important questions include how nondeterminate computations should be expressed in high-level languages, and what program execution model should guide the design of computer systems intended to support such computations.

Functional programming is uniquely capable of achieving highly parallel program execution without allowing the possibility of nondeterminate results. In this paper, we discuss how a functional programming model may be augmented to support nondeterminate computation. We begin with a brief review of past concepts and proposals for expressing nondeterminate computation. Given the hypothesis that functional programming is the best programming model for expressing determinate computation, we introduce a basic functional programming model and its implementation by means of an abstract computer system. We show how this abstract computation model may be augmented to incorporate support for nondeterminate computations by the simple addition of an *swap* memory operation. The remainder of the paper illustrates application of this base to a transaction processing example. We use the insert and search operations for integer sets to show how transactions, including multiple

[1] Imperative languages such as Fortran 90 that include support for the data parallel programming model provide the ability to write a limited class of parallel programs with a guarantee (by the compiler) of determinate results.

updates, may be permitted to overlap in time with a guarantee of correct results, and yet be written in the functional style. We compare our proposal with solutions using M-structures, as incorporated into the Ph language, and with solutions for a conventional shared-memory programming environment.

2 Background

The study of nondeterminate computation goes back many years. Here we review some of the work that has led to the proposals discussed below, but first we define nondeterminacy and explain our use of the terms *determinate* and *determinacy*.

2.1 Determinate and Nondeterminate Computation

In this paper, and in our past work on this topic, we use the terms nondeterminate and nondeterminacy rather than nondeterministic and nondeterminism. This is because there is an important distinction to be made. Determinacy is a property of the observable behavior of a system or module, whereas deterministic is generally used to describe the internal mechanism of a system (a state machine, for example). Usually, a system is called deterministic if it is characterized by a state-transition system in which each state has a unique successor state for each possible input event. Otherwise the system is nondeterministic. Determinacy is a property of systems that have asynchronous behavior and are, therefore, generally nondeterministic. Such a system is, nevertheless, determinate if and only if the following condition is satisfied (We suppose the system has several input ports at which tokens carrying values are absorbed, and several output ports at which tokens are emitted):

> For any set of token sequences presented at the input ports, every run of the system will ultimately emit the identical sequences of output tokens.

In finite state machine theory, deterministic and nondeterministic classes of machines are equivalent in the sense that given a machine, one can construct a machine of the opposite type that has the same input/output behavior. This is not the case for determinate and nondeterminate program modules, even if they are finite state.

2.2 Cooperating Sequential Processes

It has long been known how to construct concurrent programs that are nevertheless determinate. In [Van Horn 66] it was shown how several processes may jointly perform "asynchronously reproducible" computation by passing access capabilities for data cells according to a simple protocol. In 1970 Patil showed that finite interconnections of modules that pass message packets to one another and follow simple rules of behavior may be guaranteed to be determinate. Kahn showed the same property for recursive constructions over a basis of simple functional modules [Kahn 77]. These ideas were applied in the Philips RC4000 experimental operating system. Karp and Miller studied determinacy in a simple graph model of computation in [Karp 66], which is the earliest known publication on "determinacy". The determinacy of a general class of dataflow program graphs was established in [Rodriguez 67].

In [Dijkstra 68] the well-known P and V commands for process synchronization using semaphores were introduced as a device that allows more careful reasoning about the behavior of programs executed by concurrent processes. One important program structure concept introduced by Dijkstra is the *critical section*, a portion of a sequential program that must be executed by only one process at a time. He showed how the mutual exclusion requirement of critical sections can be implemented using the semaphore commands.

2.3 Transaction Handling

In the construction of static dataflow graphs [Dennis 72], it was recognized that the conditional construction required a merge actor to allow value-carrying tokens from either arm to pass to the output arc. To avoid hazards arising from the possibility that new data might overtake earlier tokens, it was necessary that operation of the merge actor be controlled by the decider of the conditional. This control feature changed the behavior of the merge actor from nondeterminate to determinate. It was discovered that many interesting nondeterminate systems (programs) could be constructed using the nondeterminate merge actor, suggesting that the nondeterminate merge might be a universal building block for nondeterminate programs. In [Dennis 76] one of the authors showed how a simple transaction processing example could be implemented using nondeterminate merge actors to funnel transaction requests from several sources to a common (functional) processing module, and tagging requests so the responses can be distributed to corresponding requesters. An open question was how a transaction server should be programmed such that arbitrarily many sources of requests could be served.

2.4 Monitors, Guardians, and Managers

Deriving form Dijkstra's concept of critical section, Hoare and Brinch Hansen evolved a programming construct called a *monitor* [Hoare 74, Brinch Hansen 93], intended to be a general scheme for encapsulating critical sections that provide mutual exclusion of access to shared data. Since the monitor was designed for a "cooperating sequential processing" environment, finding a desirable semantics for an encapsulation construct is made difficult by the well-known defects of imperative style—side effects and data races. Two proposals framed for the functional programming environment have been offered. In [Arvind 84] a *manager* construct is described that provides the same sort of encapsulation of program units for shared data in a functional programming environment. More recently *M-structures*, an extension of I-structures, have been studied as a possible universal basis for writing nondeterminate programs [Barth 92], and features in support of M-structure have been included in the (quasi-) functional programming language Ph. The discussion in this paper of using the `SWAP` memory operation as a universal basis for nondeterminate programs grew out of unpublished work completed in 1981 [Dennis 81]. The transaction processing example in this paper shows how `SWAP` can be used to implement a manager-like construct that permits concurrent execution of transactions while retaining the advantages of the functional style in programming the transaction operations.

```
data Set =
   EMPTY |
   NONEMPTY Int Set

Insert :: Set -> Int -> Set

Insert set n =
  case set of
    EMPTY -> NONEMPTY n (EMPTY)
    NONEMPTY m rest ->
      if m > n then
        NONEMPTY n set
      else
        NONEMPTY m (Insert n rest)
```

Figure 1: The `Insert` function written in Haskell.

3 Dataflow Signal Graphs

We illustrate dataflow signal graphs using an example. Figure 1 shows a function expressed in Haskell [Hudak 91] that inserts an integer into an ordered list. Figure 2 shows the `Insert` function represented as a *dataflow program signal graph*. Each actor (denoted by a box) specifies an *action* that occurs after all predecessor boxes (or the *start* node) have completed their actions. A directed arc (`u`,`v`) between two actors `u` and `v` in the graph denotes that when `u` finishes its action, a signal is to be sent to node `v` to signal its completion. Therefore, these arcs are called *signal arcs*.

An action is an operation that reads values of input variables, performs computation, and possibly assigns results to an output variable. Some boxes, drawn as oblongs, perform tests on input variables and signal successor boxes depending on outcomes of the tests. Application of a function causes: new instances of the function's variables to be created and initialized with the undefined value; argument values to be made available to the new activation; and the *start* node of the called function graph to be enabled. When the application terminates, the result value is assigned to a specified variable of the caller. Each variable must be assigned a unique and unambiguous value (or not be assigned or referenced) in any instantiation of the program graph. So that this is true, all data dependences between variables in a program graph must be represented by the partial order defined by the signal arcs of the graph.

Incremental arrays are implemented using I-structure operations [Arvind 89]. The `CREATE n` operation creates (as a heap node) an array of n elements indexed by integers $0, ..., n-1$ in which each element has the value `UNDEF`, meaning undefined. The `CREATE` operation yields a *pointer* to the heap node that may be held by a variable, stored as an array element, and used to store and access elements of the array. An action `A[i]:v` defines element `i` of array `A` to have the value `v`. Any reference `A[i]` to an array element completes (yielding the element value) only when the element has become defined.

Such Haskell programs and the dataflow signal graphs derived from them are *determinate*—the result of any function evaluation is independent of the order in which enabled nodes of the graphs are chosen for execution. In the next section we describe an *abstract computer system* capable of performing computations specified by dataflow signal graphs. Then we show how the system may be augmented and programmed to implement nondeterminate transaction processing computations.

4 The Abstract Machine Model

A shared-memory multiprocessor consists of a collection of processors and a distributed memory system organized so that each processor may access any object held in the memory. We view this general architecture according to the *abstract computer system* model shown in Figure 3. In this model, the processors interact with the memory system by presenting *commands*, and the memory responds by storing information or returning *responses* containing values of stored data. The Interprocessor Network supports function application as discussed below. In this section we describe an architecture-independent memory model matched to our dataflow program execution model.

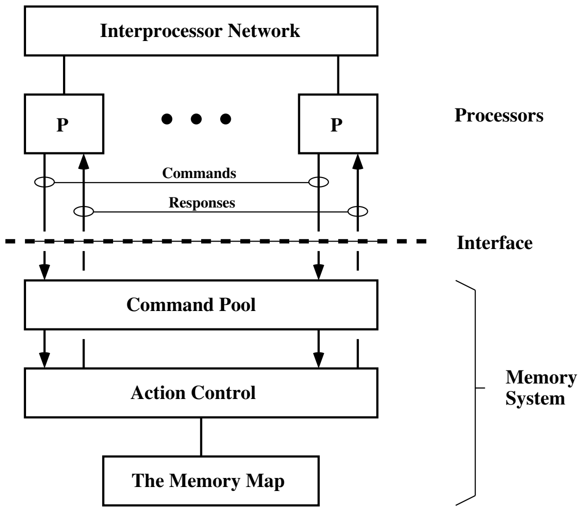

Figure 3: Abstract computer system with memory system model.

4.1 The Processors

Before discussing characteristics of the memory system, let us consider the nature of the processors. For simplicity we assume that any one function activation is performed entirely on one processor; that is, the instruction instances corresponding to one instantiation of a dataflow signal graph are all executed by the same processor. This choice has been made in most, if not all, of the experimental multiprocessors based on dataflow principles.

Given this choice, it makes sense to allocate memory for the data frames of function activations in the processors executing them. Then all references to variables held in data frames will be local memory references, and may be performed independently of the distributed memory system.

Function application is initiated by an `APPLY` instruction that starts the following sequence of events:

1. A processor is chosen for executing the new function activation.

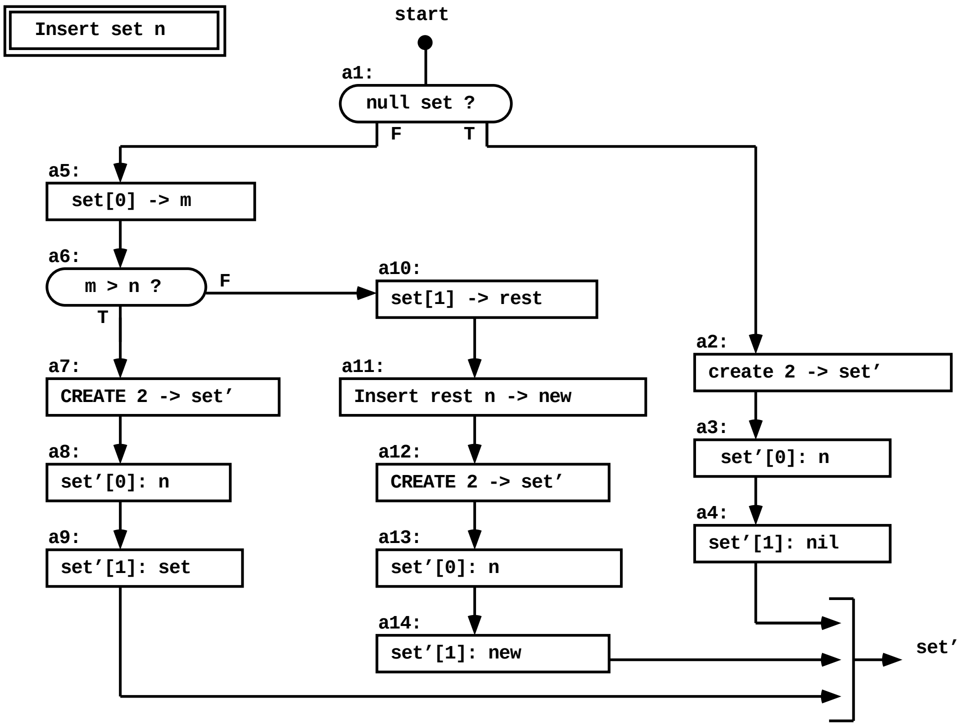

Figure 2: Dataflow program graph for the `Insert` function.

2. A memory segment is allocated at the chosen processor for the data frame of the new activation, and each cell of the data frame is set to `UNDEF`.

3. A return *continuation* consisting of the data frame address and the list of successors of the `APPLY` instruction is stored in the data frame of the new activation.

4. The function argument value is placed in the argument location of the data frame, and the start node is enabled for execution.

5. When a return node of the function is executed, the result value of the function is delivered according to the successor list.

If a function has more than one argument or result value, the argument or result values are made components of a (heterogeneous) incremental array. This choice provides lenient evaluation semantics. We assume that function initiation and termination are performed using messages sent over the Interprocessor Network.

4.2 The Memory System

We wish to specify a memory system suitable for supporting the nested function program execution model introduced above. We insist that correct operation of the system as a whole not depend on immediate responses to processors from the memory system. To enforce this requirement, the memory system model in Figure 3 includes a buffer (the Command Pool) for commands that have been presented by processors, but have not yet been acted on by the memory system. The memory system acts upon commands chosen arbitrarily (subject to fairness) from the Command Pool. Thus any criterion of correct behavior must recognize that commands may be arbitrarily delayed. (This is essentially the network delay assumption for scalable multiprocessors.)

The basic memory operations are `READ` and `WRITE`. For the purpose of supporting execution of dataflow graphs, it is helpful to view the memory system as a collection of locations where values may be placed by one computing entity for use by other entities that act concurrently. The `WRITE` command is used to place values in memory locations and the `READ` command is used to retrieve them. Now, even if one could guarantee that each `READ` is presented to the memory system only after presentation of the corresponding `WRITE`, it would be impossible to ensure that the memory system acts on the `WRITE` before acting on the `READ`. This is because either command may remain in the Command Pool arbitrarily long before being selected for action. For this reason we specify the memory system so that a pair of `READ` and `WRITE` operations on the same memory location has the same effect regardless of which operation is acted upon first. This requires that they be "synchronizing" shared-memory operations, which is just what is needed to support I-structure operations.

Here we give a specification in the Haskell language of the Action Control module of the memory system. The command messages sent to the memory have the following formats:

```
data Request =
   READ Address Node Continuation  |
   WRITE Value Address
```

In these formats the data type Address is the set of (global) memory locations, Value is the set of possible contents of memory locations, Node is the set of processor identifiers, and an entity of type Continuation contains information that identifies a specific activity to be continued by the requesting processor upon completion of the command. For our program model a continuation would consist of the address of a data frame and the offset of the instruction to be activated within the program segment of the function. (The address of the program segment could be included in the continuation, but it is usually preferable to retrieve it from the data frame.)

Messages sent by the memory system to processors in response to READ commands have the format:

```
data Message =
   (MESSAGE Node Response)
data Response =
   REPLY Value Continuation
```

The state of the memory system consists of a pool of Request messages that have been presented by processors to the memory system, but have not been acted upon, and a mapping from addresses to memory items:

```
data MemoryState :: ([Request], MemoryMap)
data MemoryMap :: Int -> MemoryItem
```

Items are of the following kinds:

```
data MemoryItem =
   (UNDEF)          |
   (DEFINED Value)  |
   (QUEUE [Entry])

data Entry =
   (ENTRY Node Continuation Entry) |
   (EMPTY)
```

The memory system acts on request messages according to the rules given in Figure 4. A READ request contains the address of the location to be read, the processor identifier, and a continuation that specifies the instruction instance that should receive the value read. If the state of the location is DEFINED, the value is sent in a response to the processor. If the state is undefined (UNDEF), the READ request has to be the first read access to the location. In this case it creates a queue with the node and continuation in its first entry. If the state of the location contains a queue, it just adds a new entry to the queue. A WRITE request contains a value and the address of the location where the value is to be written. If the state of the location is undefined (UNDEF), it writes the value into the location and changes the state to defined (DEFINED). If the state is a queue, it writes the value into the location and sends a response message for each entry in the queue. If the state is found to be defined (DEFINED), it signals an error.

5 The Swap Operation

We have found that an operation that atomically writes a new value into a location and returns the previous value held by the location can be used to implement typical instances of nondeterminate action in computations. We have named this operation SWAP and the location it references is called a *guard*. The SWAP operation is requested by the following message to the memory system

```
MemoryAction :: Request -> MemoryMap ->
                         ([Message], MemoryMap)
MemoryAction (READ address node continuation) map =
  let item = map address
  in case item of
      (UNDEF)  -> let
         q = (ENTRY node continuation (EMPTY))
         map' = MapUpdate map address (QUEUE q)
        in ([], map')
      (QUEUE q) -> let
         q' = AddToQueue node continuation q
         map' = MapUpdate map address (QUEUE q')
        in ([], map')
      (DEFINED value) ->
          (SendReply value node continuation map)
MemoryAction (WRITE value address) map =
  let item = map address
  in case item of
      (UNDEF) -> let
         map' =
             MapUpdate map address (DEFINED value)
        in ([], map')
      (QUEUE queue) ->let
         msgs = MakeResponses queue node value
         map' =
             MapUpdate map address (DEFINED value)
        in (msgs, map')
      (DEFINED value) -> ERROR

SendReply value node continuation map = let
     content = (REPLY value continuation)
     msg = (MESSAGE node content)
   in ([msg], map)
MakeResponses queue node value =
  case queue of
    [] -> []
    [(ENTRY node continuation):queue'] -> let
        content = (REPLY value continuation)
        msg = (MESSAGE node content)
      in [msg:(MakeResponses queue' node value)
```

Figure 4: The memory transition rules for the abstract computer system. In Haskell the prime character (') is valid in identifiers and primed identifiers are often used to denote the modified value of an object. Note that map denotes a function that maps locations into items, so map location evaluates to the memory item of interest. The function MapUpdate yields a new mapping in which the specified location contains the specified item. Code for the auxiliary functions SendReply and MakeResponses is included.

```
(SWAP value address processor continuation)
```

and is answered by the same format of reply as for the READ operation. The effect of the SWAP operation is specified by the addition to the memory state-transition function given in Figure 5.

The effect of executing the SWAP operation is to substitute a new value at the specified location and return the old value to the requesting processor. Note that, when no confusion may occur, we may omit the "processor" and "continuation" fields in specifying the swap operation in a textual program. The default is the processor requesting the swap and the continuation consisting of the data frame of the current activation and the instruction index of the immediately following action in the program text.

```
(SWAP value address node continuation)  ->
   let item = map [address]
   in case item of
      (UNDEF) | (QUEUE _) -> ERROR
      (DEFINED old_value -> let
            map' =
                MapUpdate map address (DEFINED value)
         in SendReply old_value node continuation map'
```

Figure 5: Memory transition rule for the SWAP operation.

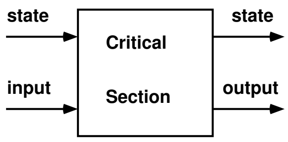

Figure 6: A critical section modelled as a state-modifying function.

A location accessed by a SWAP operation is called a *guard* for the way it may be used to implement a critical section as discussed below. In general, a guard is associated with and controls access to some resource. The address of a guard may be freely passed to any processes that needs to access the resource associated with the guard.

In our abstract computer system implementation we assume that any memory location used by a SWAP operation is never accessed using any other operation (that is, READ or WRITE). A request message for a SWAP operation is sent directly to the home memory unit for the specified location. No entry in cache memory is ever made for a guard location. A SWAP is a long latency operation, so its invocation from a thread specifies a continuation, and the processor terminates thread execution.

6 Critical Sections in Functional Programming

In conventional programming environments, a *critical section* is a portion of a program that must be entered by only one process at a time because each activation of it performs an update of shared data. Considering that we are interested in critical sections within the context of functional programs, we model a critical section as function that takes a value of type State and a value of type Input and yields a new state and a value of type Output

```
function CritSect :: State -> Input ->
                                  [State, Output]
```

as shown in Figure 6.

Two or more processes may wish to invoke the critical section at arbitrary points in the progress of their respective computations. The desired effect is illustrated in Figure 7. Note that the order in which instantiations of the critical section are placed in the composition of critical section functions is nondeterminate.

It is necessary that the critical section pass through information that will allow each output value to be directed to the correct process so it may continue its activity. We accomplish this by letting each input value be a record

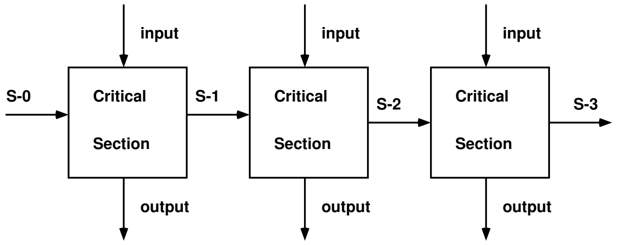

Figure 7: The effect of interleaved invocations of a critical section.

```
data Input :: (INPUT Value Continuation)
```

where a value of type Continuation represents the point from which activity by the invoking process should be continued with the output value from the critical section. Similarly, each output value is a record

```
data Output :: (OUTPUT Value Continuation)
```

and we require that any critical section function preserve the continuation from input to output.

We wish to be able to invoke the critical section code from any point in a main program by writing something like

```
output = Transact input crit_sect_funct
```

at each point where a transaction may be needed. This would mean that a position in the cascade composition of applications of the function should be reserved for the input and the corresponding output value returned. This form cannot work because, for one thing, the initial value of the state has not been specified. In addition, our program might require more than one series of transactions, each defining its own private sequence of state values, but each series using the same critical section function. What is needed is an entity that uniquely represents the particular sequence of state values of interest and associates some critical section function with the series. We call this entity a *guardian*.

We will define two generic functions for creating and utilizing guardians. The first is MakeGuardian which returns a guardian when given an initial state value and the critical section function as arguments. The second, Transact, is not really a function because, as we will show, it does the dirty work of nondeterminately inserting an instance of the critical section function into the cascade composition. Thus a main program for performing a series of transaction using CritSect as the critical section function may be written as follows:

```
Main = let
  guardian = MakeGuardian init-state CritSect
    ---
  output1 = Transact input1 guardian
    ---
  output2 = Transact input2 guardian
    ---
    ---
```

The use of a guard location and the SWAP operation are the key to implementing these two generic modules. The MakeGuardian function may be coded as follows:

```
Process :: State -> Queue ->
       (Input -> State (Output, State) -> _

data Queue = (ENTRY Input Queue)

data Input = (INPUT Value Continuation)
data Output = (OUTPUT Value Continuation)

Process state queue crit_sect_funct =
      (ENTRY input queue') = queue;
      (output, state') = crit_sect_funct input state;
      (OUTPUT result cont) = output;
            _ = continue cont result
   in
      Process state' queue crit_sect_funct
```

Figure 8: The Process function.

```
MakeGuardian init-state CritSect = let
    queue = (ENTRY undefined undefined)
    guardian = guard queue;
    _ = Process  init_state queue CritSect
  in
    guardian
```

The special operation `guard` creates a guard, fills it with the initial value `init-state`, and returns a reference to it (its address). The `MakeGuardian` function also starts the processing of the (initially empty) queue by invoking the `Process` module.

In the code language of the abstract computer system discussed in section 4, this would translate to:

```
MakeGuardian init-state crit_sect_ptr;
  queue_ptr = ALLOCATE (2);
  WRITE queue_ptr + 0, UNDEF;
  WRITE queue_ptr + 1, UNDEF;

  guardian_ptr = ALLOCATE (1);
  () = SWAP guardian_ptr queue_ptr;

  fork
     Process init_state queue_ptr, crit_sect_ptr;

  return guardian_ptr;
```

In this code the `SWAP` operation is used just to put an initial value in the guard location. The previous value in the guard location is garbage and is thrown away.

The guard location holds a pointer to the final (unfilled) entry of a queue of entries of the form

```
data Queue =
   (ENTRY Input Queue) |
   (EMPTY)
```

The generic `Process` function applies the critical section function to input values provided by the queue entries, returning the result values as specified by the given continuation in each instance. Figure 8 shows the coding of the `Process` function. (Its normal return value is empty because the effect of executing `Process` is to provide return values to each caller of `Transact` by means of the continuations saved in the entries of the queue.)

The special module `Transact` processes each call making a new entry in the queue for processing by the `Process` function. Its code is given in Figure 9. The new queue entry consists of the argument and the return continuation and becomes the final entry of the queue. This is done by the `update` operator which is implemented using the `SWAP` memory operation as shown in Figure 10.

```
Transact argument guardian = let

    cont = return_continuation;
        -- obtain the contiuation for this call
        -- of Transact
    new = (ENTRY undefined undefined);
        -- make a new unfilled queue entry
    current = update guardian new;
        -- set the guard so future requests use
        -- the new queue entry
    input = (INPUT argument cont);
        -- construct the new queue entry
    define current (ENTRY input new)
        -- fill the current queue entrye
  in _
```

Figure 9: The special module Transact.

```
Transact argument guard_ptr return_continuation

  cont = return_continuation;
  new_ptr = Allocate (2);
  WRITE new_ptr + 0, UNDEF;
  WRITE new_ptr + 1, UNDEF;
  current_ptr = SWAP guard_ptr, new_ptr;
  input_ptr = Allocate (2);
  WRITE input_ptr + 0, argument;
  WRITE input_ptr + 1, cont;
  WRITE current_ptr + 0, input_ptr;
  WRITE current_ptr + 1, new_ptr;
  QUIT;

end Transact
```

Figure 10: Abstract machine code for the Transact module.

The way this code appends entries to the queue for the `Process` function is illustrated by the succession of snapshots in Figure 11. In Figure 11(A) the guard holds a reference to current (and final) entry `b` of the queue. The `Process` function is following the chain of entries from the top and pauses when it attempts to access (with a `READ` operation) entry `b` which is yet to be filled in. In Figure 11(B), a new (empty) entry `c` has been created and a `SWAP` operation performed to substitute `c` for `b` in the guard. The new empty entry is now referenced by the guard. In Figure 11(C) the pointer to `c` has been installed (by a `WRITE` operation) in the current entry `b`, allowing the `Process` function to proceed.

7 An Example: Insertion and Search in an Integer Set

A simple example for transaction processing is the processing of commands for inserting and searching in a set of integers. Figure 12 shows the functions that, respectively, insert integers into an ordered list, and search for an integer in the list.

We define search and insert commands and the corresponding responses as follows:

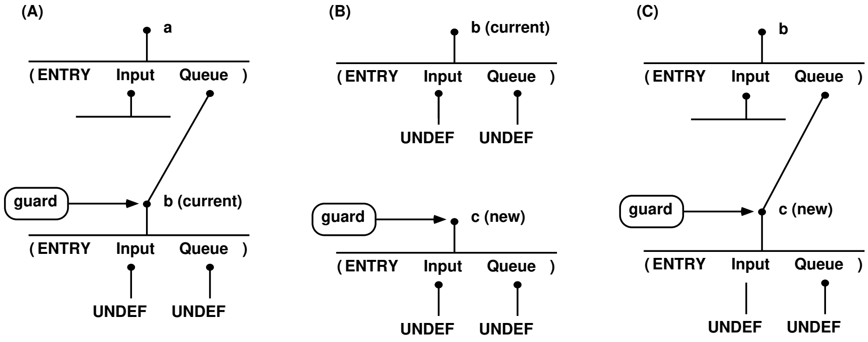

Figure 11: Snapshots showing formation of the stream of `Entries` at the guard.

```
data Set =
   EMPTY |
   NONEMPTY int Set

Insert set n :: Set -> Int -> Set
Search set n  :: Set -> Int -> Bool

Insert set n =
  case set of
    EMPTY -> NONEMPTY n (EMPTY)
    NONEMPTY m rest ->
      if m > n then
        NONEMPTY n set
      else
        NONEMPTY m (Insert n rest)

Search set n =
  case set of
    EMPTY -> FALSE
    NONEMPTY n rest -> TRUE
    NONEMPTY _ rest ->
      Search rest n
```

Figure 12: The `Insert` and `Search` functions.

```
data Command = SEARCH int | INSERT int
data Response = SEARCH bool | INSERT
```

We must provide for continuing operation of an arbitrary caller following completion of each instance of the critical section. For this purpose we include a `continuation` with the arguments and result values for the critical section function.

```
data Input = (INPUT Command Continuation)
data Output = (OUTPUT Response Continuation)
```

The `CritSect` function for this example has a set as its state and invokes the insert or search function according to the command given as input. For the search command it returns a boolean value and the same set as it was given. For the insert command it returns the given set with the given integer inserted.

```
CritSect :: Input -> Set -> (Output, Set)

CritSect input set = let
      (cmnd, cont) = input
      (resp, set') = case cmnd of
         SEARCH n -> SEARCH (Search set n), set
         INSERT n -> INSERT, (Insert set n)
      output = (resp, cont)
   in (output, set')
```

This critical section function, together with the insert and search functions, form the complete application-specific code for the integer set example. The other program modules (`MakeGuardian`, `Transact`, and `Process`) are unchanged and are common to all transaction processing implementations.

This example illustrates how the use of I_structure synchronization can yield substantial parallelism from the concurrent processing of many transactions. The `CritSect` function may return its set output component even though its activity (searching the set or scanning for the correct insertion point and constructing a new list) has not completed. Note that each entry created to hold a new element of a list representing the integer set is written only once after space is reserved by the `ALLOCATE` instruction in the insert code. The absence of updates and the delaying of access until an entry is filled combine to achieve the concurrency of operation without departing from the interleaving semantics of critical sections.

8 Other Approaches

In this section we discuss other approaches for handling transactions in a multiple processor setting. The principle issues addressed are using the clarity of the functional programming style as much as possible, and achieving a high degree of concurrency, specifically overlapping the execution of transactions. First we consider ways in which M-structures may be used to implement the integer set example, then we discuss how it might be implemented in a conventional shared-memory multiprocessor.

8.1 Programming with M-Structures

One alternative to extend functional programming for nondeterminate computation is to introduce updateable objects called *M-Structures* [Barth 92]. Every M-structure location is either *empty* or *full*. An M-structure *take* operation on an empty slot suspends. A *put* operation to an empty slot simply writes a value in the slot and makes the slot *full* if no pending *take* operations are waiting. Otherwise, the value is passed to one of the suspended *take* operations, and the slot is again *empty*. A *put* to a full slot is an error.

Below is how the integer set example can be written with M-structures. Since search is a pure function (no M-structure operation is needed), we only give code in Id for the insert operation.[2]

```
  define Insert set n =
    if (MNil? set) then
       MCons n nil
    else
      { m = set.MCons_1;
        IN
      if m > n then
      MCons n set
      else
L:      {set!MCons_2 = insert set!MCons_2 n;
       IN
          set
      }
    }
```

Note that, in line L, the '!' on the left-hand-side means M-structure *put* and on the right-hand side means an M-structure *take* operation. See [Nikhil 90] for cetails concerning I-structures in Id. Note that the M-structure *put* operation is an in-place update, so there is no copying.

To highlight the difference of the above code from our solution, consider what happens when several concurrent insert or search operations are present. In our solution, once a command (insert or search) is presented to the critical section, its effect is fully determined, and the results do not depend on the relative speed of execution of each command. The nondeterminacy is confined to the interleaving of the several commands by the operation of the `Transact` module.

The same cannot be said for the M-structure solution. In fact, if there are two concurrent insertions, say S1 and S2, it is possible for S1 and S2 to continue to leapfrog each other toward the end of the list. The result of a concurrent search operation, if present, depends on the speed of its execution relative to any concurrent insert operations.

Our solution is consistent with the usual semantics for database transaction processing. The results are the same as in the sequential run in which each transaction is executed completely before any other is allowed to start. On the other hand, when M-structure operations are used to modify parts of the shared state in place, we know of no general methodology for ensuring that the semantics of database transactions are satisfied.

It may be that our solution using a guard and functional operations performs more memory allocation operations than does the M-structure code using transaction operations written in a non-functional style. However, in our solution, additional memory is actually needed only when overlapped transactions are in progress. When such a situation arises, we suspect that the M-structure code will incur some extra cost for M-structure coordination. If this cost involves allocation steps it may as expensive as copying. Of course, one alternative is to implement our `Transact` module using an M-structure variable in place of a guardian, but we believe using the `SWAP` operation is both simpler and more efficient.

[2] We expect to use the pH language [Nikhil 94] to present this version in the final paper.

8.2 Programming Integer Set in a Shared-Memory Environment

Here we consider how multiple invocations of the insert and search operations may be handled on a cache-coherent shared-memory multiprocessor computer. We assume that all processors share a common address space, so each processor is able to execute any procedure present in the memory. We assume that processes use the standard semaphore operations P and V to implement the necessary coordination. The system must guarantee (through its cache coherence machinery) that in execution of the code sequences

```
   P1           P2

 write x       P(s)
 V(s)          read x
```

by processors P1 and P2, whenever execution of V by P1 causes execution of P by P2, the value of x read by P2 is always identical to the value written by P1. (There must be no failure due to races in the memory network.)

The simplest solution is to use a single mutual exclusion semaphore that permits one process at a time to execute the insert or search function. This has the drawback that no concurrency of transaction execution is achieved, but the advantage that only one instance of the list representing the integer set exists at any time.

Now let us consider how we might construct an implementation such that transactions could overlap as in our solution using a guardian. There are two parts to the problem. The first is entering the commands into a common stream. This part can be done using a mutual exclusion semaphore guarding a critical section that appends a new input record to a queue.

The second part is the coding of a daemon process that will yield overlapped execution of the insert and search operations. For concurrent operation of commands, we suppose a separate process is associated with the execution of each transaction. Two problems must be dealt with: (1) Making sure that each process is given the right instance of the set list. (2) Making sure that, if an insert operation is active, then each subsequent operation waits at each node until the node becomes defined.

Part (1) is done by a process that takes entries from the queue and either (a) starts a search process using the current list of set elements, or (b) initializes a new list and starts an insert process building the new list from the current one; the new list is made the current list for subsequent queue entries.

To do part (2), we need a synchronization mechanism that permits any number of search and insert operations to pause at a list node with undefined contents. These operation are allowed to continue just when the creator of the node (some earlier insert process) has defined the node contents, effectively simulating an I-structure variable.

In summary, if concurrency of transactions is not essential, transaction processing may be implemented using functional operation modules and a single mutual exclusion semaphore. However, it appears difficult to code for overlapped execution of transactions (other than multiple readers) without resorting to methods that lack the elegance and simplicity of the functional style.

9 Conclusion

We have introduced a way of programming the processing of access and update transactions in a multiprocessor computer system such that overlapped execution of multiple transactions is achieved and yet program modules for the transaction operations are written in the pure functional style. This is accomplished through the use of I-structures and the addition of a `SWAP` operation to the memory system of an abstract computer for executing pure functional programs.

Comparison with published solutions using M-structures shows that, although the published solutions offer the advantage of update-in-place in avoiding allocation steps, this is attained at the cost of departing form the pure functional style in programming the transaction operations. Programming transactions for overlapped execution in a conventional shared-memory multiprocessor appears to require programming techniques that make heavy use of process coordination commands and will be difficult to prove correct.

10 Acknowledgments

The authors would like to thank the MIT Laboratory of Computer Science and the School of Computer Science of McGill University for facilitating this effort. The second author would like to thank the National Science and Engineering Research Council (NSERC) of Canada for their support.

References

[Arvind 84] Arvind and J. D. Brock. Resource managers in functional programming. *Journal of Parallel and Distributed Computing*, 1(1):5–21, August 1984.

[Arvind 78] Arvind, Kim P. Gostelow, and Wil Plouffe. *An Asynchronous Programming Language and Computing Machine.* Technical Report 114a, Department of Information and Computer Science, University of California, Irvine, December 1978.

[Arvind 89] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM TOPLAS*, 11(4):598–632, October 1989.

[Barth 92] Paul S. Barth. Atomic data structures for parallel computing. Technical Report MIT/LCS-TR-532, Laboratory for Computer Science, MIT, March 1992.

[Brinch Hansen 93] Per Brinch Hansen. Monitors and concurrent Pascal: A personal history. In *History of Programming Languages Conference, Preprints*, pages 1–35. ACM, 1993.

[Dennis 72] Jack B. Dennis, John Fosseen, and John Linderman. Data Flow Schemas. In *International Symposium on Theoretical Programming*, no. 5 in *Lecture Notes in Computer Science*, pages 187–215. Berlin: Springer-Verlag, 1972.

[Dennis 74] Jack B. Dennis. First version of a data-flow procedure language. In *Proceedings, Colloque sur la Programmation*, no. 19 in *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, 1974.

[Dennis 76] Jack B. Dennis. A language design for structured concurrency. In *Design and Implementation of Programming Languages*, pages 231–243. Springer-Verlag, LNCS-54, 1976.

[Dennis 81] Jack B. Dennis, Data should not change: a model for a computer system. CSG Memo 209, Computation Structures Group, MIT Laboratory for Computer Science, July 1981.

[Dijkstra 68] Edsger W. Dijkstra. Communicating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. New York: Academic Press, 1968.

[Hagersten 92] Erik Hagersten, Anders Landin, and Seif Haridi. DDM—A cache-only memory architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[Hoare 74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17:549–557, October 1974.

[Hudak 91] Paul Hudak (editor), Simon Peyton Jones (editor), Philip Wadler (editor), Brian Boutel, Jon Fairbairn, Joseph Fasel, Maria M. Guzman, Kevin Hammond, John Hughes, Thomas Johnsson, Richard Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell, a non-strict purely functional language (version 1.1). Yale University, Department of Computer Science, August 1991.

[Kahn 77] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74*, pages 993–998. North-Holland, 1977.

[Karp 66] R. M. Karp and R. E. Miller. Properties of a model for parallel computation: Determinancy, termination, queueing. *SIAM Journal of Applied Math.*, 14, November 1966.

[Lenoski 90] Daniel Lenoski, Kourosh Gharachorloo, James Laudon, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. Design of scalable shared-memory multiprocessors: The DASH approach. In *COMPCON Spring '90: Digest of Papers*, pages 62–81. IEEE Computer Society, March 1990.

[Nikhil 90] Rishiyur S. Nikhil and Arvind. Id: A language with implicit parallelism. Computation Structures Group Memo 305, Laboratory for Computer Science, MIT, 1990.

[Nikhil 94] Rishiyur S. Nikhil, Arvind, James Hicks, Shail Aditya, Lennart Augustsson, Jan-Willem Maessen and Yuli Zhou. pH Language Reference Manual, Version 1.0—preliminary. September 1994.

[Rodriguez 67] Jorge E. Rodriguez. *A Graph Model for Parallel Computation.* PhD thesis, Laboratory for Computer Science, MIT, Cambridge, MA, September 1967.

[Van Horn 66] Earl C. Van Horn. *Computer Design for Asynchronously Reproducible Multiprocessing.* PhD thesis, MIT Department of Electrical Engineering, August 1966.