Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

A FUNCTIONAL APPROACH TO SIMULATION PROGRAMMING

Adrienne Bloss

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061-0106

ABSTRACT

In this paper we explore simulation programming in Haskell, a purely functional language. We show how a simple simulation problem can be modeled in Haskell, and compare the result to four traditional approaches. Functional languages are lower-level than some simulation languages, but much higher-level than traditional, general-purpose languages such as C or Pascal. We claim that for modeling at least one class of simulation problems, functional languages are better than other general-purpose languages, and may be as good as many simulation languages.

1. INTRODUCTION

Simulation is an important application of computer science, and research in simulation has made great advances in the last two decades. Most simulation programming is done in traditional special-purpose simulation languages such as GPSS, SIMAN, SIMSCRIPT, SIMULA, or SLAM. More recent simulation languages such as Maisie [Bagrodia 1990], Sim++ [Lomow 1989], ModSimII [Bryan 1989], and CPS [Abrams 1989] are designed for parallel execution. However, sometimes general-purpose languages are appropriate for simulation programming [Balci 1988]. For example, no simulation language may be available for a particular system, or the simulation model may need to be integrated with another system written in a general-purpose language. Furthermore, each simulation language is designed around a particular world view, or *conceptual framework*, and the frameworks offered by the available simulation languages may not be appropriate for the problem to be modeled. Although every general-purpose language is based on some model of computation, these models are usually more general than the world views underlying simulation languages.

In each of these cases, simulation languages are not appropriate and a general-purpose language should be used instead. Unfortunately, traditional imperative languages such as C and Pascal reflect the sequential execution of the Von Neumann architecture, and are poorly suited for modeling the concurrent logic inherent in most simulation problems. In this paper, we explore simulation programming in Haskell [Hudak et al. 1990], a purely functional language. We show how a simple simulation problem can be modeled in Haskell, and compare the result to four traditional approaches. Except for a brief discussion in an otherwise unrelated paper [Hudak and Anderson 1988], we know of no other work in this area.

In the next section, we introduce Haskell and describe the features that play the most important roles in the simulation program. In Section 3, we present the problem to be solved, and in Section 4, we develop a solution in Haskell. In Section 5, we compare the Haskell solution with traditional solutions based on event scheduling, activity scanning, three-phase approach, and process interaction. In Section 6, we present our conclusions, and in Section 7 we discuss future work.

2. AN OVERVIEW OF HASKELL

Haskell is a purely functional language that has been developed over the last two years, and is expected to serve as a standard for functional programming. A Haskell program is a collection of functions, possibly mutually recursive, and the meaning of the program is the value of the top-level identifier "main." Function application is denoted by juxtaposition, so what would be written as `f(x,y)` in most imperative languages is written `f x y` in Haskell. Unlike in LISP, parentheses are used only for grouping, and are often used redundantly for clarity. For example, the following program computes the factorial of 3:

```
factorial n acc = if n<=1 then acc
                  else factorial (n-1) (n*acc)

main = factorial 3 1
```

We will usually drop `main`, discussing only the function definitions.

Although no type declarations appear in this example, Haskell is statically typed. The programmer may specify types of functions, but may also omit them. If a function's type is not specified, the compiler will try to infer it from the function's definition. If it is unable to do so, it will generate an appropriate warning.

Many interesting and important features of Haskell are not discussed in this brief introduction. In particular, modules, polymorphism, type classes, and stream I/O greatly influence the style of programming encouraged in Haskell, but are omitted here. The interested reader should refer to the Haskell Report [Hudak et al. 1990] for more information. In the current discussion, we will concentrate on imparting a reading knowledge of the Haskell features that are important in Section 4, including lists, pattern-matching, and lazy evaluation. The reader may find it helpful to skim this section on first reading, and refer to it as necessary when reading Section 4.

2.1 Lists

Lists are the basic data structure in Haskell. Lists are delimited by square brackets, and list elements are separated by commas, e.g., `[1,2,3]`. `[]` represents the empty list. The polymorphic infix list constructor : has type $\alpha \times [\alpha] \rightarrow [\alpha]$ for any data type α, where $[\alpha]$ signifies the type "list of α". Selectors `hd` and `tl` select the first and rest of a list, respectively. Thus for an element `e` of type α and a list `l` of type $[\alpha]$, `hd(e:l) = e` and `tl(e:l) = l`.

2.2 List Comprehensions

List comprehensions provide a concise and elegant way of specifying lists, taking much of their form from set notation. We will begin with an example:

```
odds l = [x | x <- l, odd x]
```

`Odds` is a function that takes a list (of integers) `l`, and produces a new list containing only the odd elements in `l`. The list comprehension `[x | x <- l, odd x]` specifies the list of all `x` such that `x` is taken from `l` and `odd x` is true. In general, a list comprehension specifies a list whose elements are determined by evaluating the expression on the left side of the `|` using the bindings generated by evaluating the expressions on the right side of the `|` in a nested, depth-first manner. These expressions are of two types: *generators*, such as `x <- l`, which provide new bindings for identifiers, and *guards*, such as `odd x`, which restrict which of those bindings may be used. The expressions on the right side of the `|` are evaluated from left to right; if a guard fails, evaluation backtracks until it finds a generator, at which point it takes the next element generated and proceeds forward again. When all expressions have been evaluated, the left hand side is evaluated using the resulting bindings. Evaluation then backtracks to the last generator, takes the next value, and goes forward again until the end, at which point the left hand side is evaluated again, producing the second element of the list. This continues until all generators are exhausted.

2.3 Pattern Matching

Pattern matching is a syntactic sugaring that enhances readability of function definitions. A function's formal parameters may specify the structure of the actuals, and a function definition "matches" an invocation only if the structure of the actuals matches that of the formals. Consider the following definition of the standard `map` function, which applies a function to each element of a list:

```
map f l = if l==[] then []
                   else (f (hd l)):(map f (tl l))
```

Now consider the same definition, written using pattern matching:

```
map f [] = []
map f x:xs = (f x):(map f xs)
```

This shows how : can be used to destructure, as well as construct, lists. If the pattern `a:b` is bound to a list `l`, `a` is bound to `hd l` and `b` is bound to `tl l`.

When a function has multiple definitions due to pattern-matching, they are tried in order from top to bottom, and the first one to match is executed. Argument matching is attempted from left to right until all arguments have matched or one fails. As another example, `factorial` could be defined using pattern matching as follows:

```
factorial 0 acc = acc
factorial 1 acc = acc
factorial n acc = factorial (n-1) (n*acc)
```

Function application has higher precedence than : or arithmetic operators such as `*` and `-`, so the parentheses in both definitions of `map` are redundant, but the parentheses in `factorial` are necessary.

2.4 Lazy Evaluation

Like most modern functional languages, Haskell uses a lazy evaluation strategy. This means that an expression is not evaluated until its value is demanded in some greater context. One of the implications of this evaluation strategy is that infinite lists can be defined and manipulated with ease. Three examples appear below.

```
ones = 1:ones
              -- the infinite list of 1s

nums_from n = n:(nums_from (n+1))
              -- the infinite list of integers
              -- starting at n

odd_nums_from n = [ x | x <- nums_from n, odd x]
              -- the infinite list of odd integers
              -- starting at n
```

Infinite lists, or *streams*, can be manipulated like any list — they can be passed to or returned from functions, and can be destructured into a head and tail. However, if a function such as `map` tries to use *every* head, or *every* tail of an infinite list, it will compute forever. Streams are often used in defining problems where the number of elements that will finally be required is not relevant to the form of the solution. An outer call usually takes a finite prefix of an infinite list in order to produce a finite result. In the program below, the use of `prefix` in `main` ensures that only ten elements of `squares` will be produced:

```
nums = nums_from 1

prefix n [] = []
prefix 0 l  = []
prefix n x:xs = x:(prefix (n-1) xs)

square x = x * x

squares =  map square nums

main = prefix 10 squares
```

3. A SAMPLE SIMULATION PROBLEM

The problem used to illustrate simulation programming in Haskell is taken from Balci's [1988] paper. Briefly, it describes the behavior of a multiple virtual storage batch computer system with two CPUs and a printer, and with jobs entering from four different sources. The problem is described in detail below.

A multiple virtual storage (MVS) batch computer system operates with two central processing units (CPUs). Users submit their batch programs to the MVS by using the *submit* command on an interactive virtual memory (VM) computer system running under the CMS operating system. The users of MVS via VM/CMS are classified into four categories: (1) users dialed in by using a modem with 300 baud rate, (2) users dialed in by using a modem with 1200 baud rate, (3) users dialed in by using a modem with 2400 baud rate, and (4) users connect to the local area network (LAN) with 9600 baud rate. Each user develops a batch program on the VM/CMS computer system and submits it to the MVS for processing. Assume that the interarrival times of batch programs to the MVS with respect to each user type are determined to have an exponential probability distribution with means of 3200, 640, 1600, and 266.67 seconds for the 300, 1200, 2400, and 9600 baud users respectively.

A batch program submitted first goes to the job entry subsystem (JES) of MVS. The JES scheduler (JESS) assigns the program to processor 1 (CPU1) with a probability of 0.6 or to processor 2 (CPU2) with a probability of 0.4. At the completion of program execution on a CPU, the program's output is sent to the user's virtual reader on the VM/CMS with a probability of 0.2 or to the printer (PRT) with a probability of 0.8. Assume that all queues in the MVS computer system are handled by a first-come-first-served discipline, and that each facility (JESS, CPU1, CPU2, and PRT) processes programs one at a time. The processing times of these facilities also have exponential distributions, with respective means of 112, 226.67, 300, and 160 seconds.

Assuming that the simulation model reaches the steady-state condition after 3,000 programs, simulate the system for 15,000 programs in steady state and construct confidence intervals for the following performance measures:

1. Utilization of the JESS.
2. Utilization of CPU1.
3. Utilization of CPU2.
4. Utilization of the printer.
5. Average time spent by a batch program in the MVS system.
6. Average number of batch programs in the MVS system.

4. A FUNCTIONAL APPROACH

In this section we develop a Haskell solution for the problem defined above. First we show how to view random numbers functionally, and generate distributions for the arrival times of incoming jobs, CPU and output assignments, and processing times for the CPUs and printer. We then show how the behavior of each component of the system is modeled, including the JESS, the CPUs, and the printer. Finally, we show how to compute the performance measures described above.

4.1 Preliminaries

The key structure in the functional approach is the infinite list, or stream. There are conceptually a number of queues associated with this problem, and it is natural to model these queues with streams, but streams serve other purposes as well. First, we define a stream of uniformly distributed pseudo-random numbers called `Rand`. This stream may be created by using a multiplicative congruential random number generator, where the i^{th} random number depends on the $(i-1)^{st}$ number recursively, e.g., $r_i = a * r_{i-1} \ mod \ m$ for some values of a and m. Given an initial value `r0`, `Rand` is defined using a list comprehension as follows:

```
Rand = r0 : [a*r mod m | r <- Rand]
```

This definition should be read as follows: `Rand` is the list whose head is `r0`, and whose tail is the list whose $i+1^{st}$ element is $a *$ `rand`$_i$ $mod \ m$, where `rand`$_i$ is the i^{th} element of `Rand`. Thus `Rand = [r0, a*r0 mod m, a*(a*r0 mod m) mod m, ...]`. An arbitrary number of independent streams of random numbers can be defined by parameterizing `Rand` with respect to `r0`, but for ease of presentation we will draw from only one such stream.

Given this stream of random numbers, a stream of exponentially distributed numbers with mean `lam` is easily defined in the usual way:

```
Expdist lam = [-(ln r)/lam | r <- Rand]
```

Of course, the elements of this list may be incrementally summed to give the arrival time of each item:

```
Cumul_expdist lam =
     r0 : [x + -(ln r)/lam | x <- Cumul_expdist lam,
                             r <- Rand]
```

Given these tools, we can define the streams of times at which jobs enter the MVS:

```
M300Dist  = Cumul_expdist 3200
M1200Dist = Cumul_expdist 640
M2400Dist = Cumul_expdist 1600
LANDist   = Cumul_expdist 266.67
```

We would like to merge these four queues into one, the queue of jobs entering the system. To do this, we define a `Merge` function that merges two queues of times:

```
Merge xs [] = xs
Merge [] ys = ys
Merge x:xs y:ys = if x < y then x:(Merge xs y:ys)
                           else y:(Merge x:xs ys)
```

To merge the four queues above, we simply apply nested calls to `Merge`:

```
InitDist =
    Merge M300Dist
          (Merge M1200Dist
                 (Merge M2400Dist LANDist))
```

Queues of the JESS, CPU, and printer processing times may be computed in the same way:

```
JESSDist    = Cumul_expdist 112
CPU1Dist    = Cumul_expdist 226.67
CPU2Dist    = Cumul_expdist 300
PrinterDist = Cumul_expdist 160
```

Finally, we need to represent the probability that a job will go to CPU1 or CPU2, and the probability that it will go the printer. `CPUDist` is a stream of 0s and 1s, where 1 indicates a job to be executed on CPU1, and 0 indicates indicates a job to be executed on CPU2. `OutDist` is similar, with 1 indicating a job to be printed.

```
filter rand limit = if rand < limit then 0 else 1

CPUDist = [filter r 0.4 | r <- Rand]

OutDist = [filter r 0.2 | r <- Rand]
```

4.2 Modeling the JESS

The first stop for a job is the JESS. A job enters the JESS when all of the following conditions hold:

1. The job has entered the system.
2. All jobs before it have entered the JESS.
3. The JESS is free.

Jobs entering the JESS are modeled by the stream `EnterJESS`. `EnterJESS` is based on `InitDist` and the stream of times at which the JESS is available, which is defined below as `JESS`.

```
EnterJESS = [max init jess_avail |
                    init       <- InitDist,
                    jess_avail <- JESS]
```

This definition may be read as follows: The next job will enter the JESS at the maximum of the time at which the job enters the system and the time at which the JESS is next available.

The stream of jobs leaving the JESS depends on the jobs entering the JESS and the time it takes the JESS to process each job:

```
LeaveJESS = [enter + jess_proc |
                    enter     <- EnterJESS,
                    jess_proc <- JESSDist]
```

Finally, the availability of the JESS depends on the time at which it finishes processing each job. It is also available at time zero, so we cons 0 onto the front of the stream of available times:

```
JESS =  0 : LeaveJESS
```

4.3 Modeling the CPUs

Once a job leaves the JESS, it waits to be executed by the appropriate CPU. The stream of jobs entering each CPU depends on the stream of jobs leaving the JESS, the CPU scheduling distribution, and the availability of the CPU. First we define a new stream, `Job&CPU_pairs`, in which we pair the elements of `LeaveJESS` and `CPUDist`. `Job&CPU_pairs` is used in `EnterCPU1` to select only the jobs that are scheduled for CPU1.

```
Job&CPU_pairs = [ [t,cpu] | t   <- LeaveJESS,
                            cpu <- CPUDist]

EnterCPU1 = [max leave_jess cpu1_avail |
                 [leave_jess, cpu] <- Job&CPU_pairs,
                  cpu = 1,
                  cpu1_avail <- CPU1]
```

As for the JESS, the stream of jobs leaving each CPU depends on the stream of jobs entering it and the processing time required for each job. The availability of the CPU depends on the stream of jobs leaving it.

```
LeaveCPU1 = [enter + cpu1_proc |
                    enter     <- EnterCPU1,
                    cpu1_proc <- CPU1Dist]

CPU1 = 0 : LeaveCPU1
```

A similar characterization holds for CPU2.

4.4 Printing and Leaving the System

After leaving the CPU, some jobs are printed, while others go directly to the user's output device. `OutDist` contains a 1 for each job to be printed and a 0 for each job not to be printed. Since printing and non-printing jobs may come from either CPU, we need to merge the streams leaving the CPUs to get the stream of jobs to print. As when modeling the CPUs, we pair the elements of this stream with the output distribution before defining the jobs entering the printer.

```
Job&Out_pairs = [ [t,out] |
                  t   <- Merge LeaveCPU1 LeaveCPU2,
                  out <- OutDist]

EnterP = [max leave_cpu printer_avail |
                  [leave_cpu, out] <- Job&Out_pairs,
                  out = 1,
                  printer_avail <- PRINTER]

LeaveP = [enter + print_time |
                  enter      <- EnterP,
                  print_time <- PrintDist]

PRINTER = 0 : LeaveP
```

Finally, the times of the jobs that will not be printed comprise JobsNoPrint, and the stream of system departure times is the merge of the streams of times of non-printing jobs leaving the CPU and printing jobs leaving the printer:

```
JobsNoPrint =
  [ t | [t,out] <- Job&Out_pairs, out = 0]

LeaveSys = Merge LeaveP JobsNoPrint
```

4.5 Performance Measures

The problem outlined in Section 3 states that the simulation model reaches the steady-state condition after 3,000 programs, and requires that a number of performance measures be evaluated for 15,000 programs in steady state. Let `SS_length` be the number of steady state programs to be sampled, and `Trans_length` be the number of programs that must go through the system before the steady state is reached. `Nth_tail` simply finds the n^{th} tail of a list, and `Interval` takes two integers `i` and `n` and a list `l` and returns the list of the `n` elements starting at the i^{th} element of `l`. `Sample` takes a stream and returns the elements of that stream that comprise our sample.

```
SS_length = 15000

Trans_length = 3000

Nth_tail 0 list = list
Nth_tail n x:xs = Nth_tail (n-1) xs

Interval i n l = prefix n (Nth_tail (i-1) l)

Sample l = Interval Trans_length SS_length l
```

In addition, the total time spanned by the processing of these 15,000 jobs is a useful piece of information:

```
Entry_times = Sample InitDist

Exit_times  = Sample LeaveSys

First_in    = hd Entry_times

Last_out    = hd (Nth_tail (SS_length - 1) Exit_times)

Total_time  = Last_out - First_in
```

We now address the performance measures one at a time.

1. *Utilization of the JESS.* The utilization of a resource is the total time the resource is busy divided by the total system time. For the JESS, this is easily computed by summing over the stream of JESS processing times and dividing by the total time:

```
Sum_list [] = 0
Sum_list x:xs = x + Sum_list xs

JESS_util =
  (Sum_list (Sample JESSDist)) / Total_time
```

2. *Utilization of the CPUs.* Utilization of the CPUs can be determined in much the same way as utilization of the JESS, but only the jobs that are executed on the specified CPU are in each list of CPU intervals. Also, the number of transient and steady-state jobs for each CPU must be computed. Recall that `CPUDist` contains a 1 for each CPU1 job and a 0 for each CPU2 job.

```
CPU1_trans =
  Sum_list (prefix Trans_length CPUDist)

CPU1_SS = Sum_list (Sample CPUDist)

CPU1_times =
  Interval CPU1_trans CPU1_SS CPU1Dist

CPU1_util  = (Sum_list CPU1_times) / Total_time
```

Utilization for CPU2 is determined in a similar manner.

3. *Utilization of the printer.* Again, only the jobs that are scheduled for printing are in the list of printer intervals.

```
Print_trans =
  Sum_list (prefix Trans_length OutDist)

Print_SS    = Sum_list (Sample OutDist)

Print_times =
  Interval Print_trans Print_SS PrintDIST

Print1_util = (Sum_list Print_times) / Total_time
```

4. *Average time spent by a batch program in the MVS system.* This can be computed by first computing the area under the curve representing the number of programs in the system at each time t. We compute this area by computing, for each job arrival and departure, the product of the number of jobs in the system and the time interval immediately preceding the arrival or departure. These values are summed for the total area under the curve. In function `area`, `jobs` is the current number of jobs in the system, `t` is the time at which the area was last computed, `e:es` is the list of entry times, and `d:ds` is the list of departure times.

```
Area jobs t [] [] = 0
Area jobs t e:es l:ls =  if e < l
     then (e-t)*jobs + (Area (jobs + 1) e es
     else (l-t)*jobs + (Area (jobs - 1) l e:es ls)

Total_area =
  Area 0 First_in Entry_times Exit_times
```

The average time spent in the system is simply the area divided by the number of jobs:

```
Avg_time_in_sys = Total_area / Steady_state_length
```

5. *Average number of batch programs in the MVS system.* This can also be computed using the area defined above. In this case, the area is divided by the total system time.

```
Avg_jobs_in_sys = Total_area / Total_time
```

5. COMPARISON TO TRADITIONAL APPROACH

We have shown how a simple simulation problem can be specified using a purely functional language. How does this solution compare to a solution in a traditional general-purpose language? The main difference is the absence of an explicit time-flow mechanism in the functional solution. Time flow is an integral part of each of the world views used in traditional simulation programming, and must be programmed explicitly in most general-purpose languages. Another difference is that the functional solution is more easily developed and understood in pieces, without knowing the exact context in which each piece is used. This modularity supports the use of modern software engineering techniques. Finally, each world view has its own differences from the functional model; these differences are discussed briefly below.

Event scheduling is perhaps the most different from the functional approach, and also the lowest-level of the traditional approaches. The central event queue inhibits a descriptive solution, and suggests a step-by-step execution of the simulation. *Activity scanning* is a higher-level approach, as the activities and their associated conditions can be defined declaratively, but the machinery to scan the activities and manage the time-flow mechanism can obscure the overall program structure. In a simulation language, this machinery would be embedded in the system, and the solution would be much closer to the functional solution. Activity scanning is often implemented with the *three-phase approach*, which sacrifices logical structure for efficiency by separating out solely time-dependent actvities. This is clearly a step away from the functional model. Perhaps the conceptual framework closest to the functional approach is *process interaction*. Here, objects in the system are divided into static and dynamic, and the dynamic objects are moved through the system, consisting of static objects, as conditions for their motion permit. The dynamic objects correspond to the times in the `Enter_` and `Leave_` queues in our example, and the static objects correspond to the queues associated with availability of the JESS and other resources. However, in process interaction the dynamic objects are *explicitly* moved through the system, and are classified by execution status instead of by their logical role in the system.

6. CONCLUSIONS

We have presented an elegant way of modeling a simulation problem using the lazy functional language Haskell. We have shown that infinite lists, or streams, provide a natural way of modeling queues, and that the properties of a simple queueing problem can be expressed declaratively by specifying the interrelationships between queues. Lazy evaluation allows the functional solution to avoid the time-flow mechanism used in most simulation frameworks, and we argue that the resulting solution is simpler and more intuitive. As a result, we believe that functional languages are preferable to traditional high-level languages for modeling some classes of simulation problems. Although we have not explored the relationship between functional languages and simulation languages, we suspect that for at least some problems, functional languages will offer elegant and concise solutions as well as a more general framework.

This paper has provided only a brief look at the possible role of functional programming in simulation, and much further study is required. Two of the most important remaining issues are discussed in the next section.

7. FUTURE RESEARCH

Two important issues remain to be addressed:

1. Is the functional approach appropriate for all types of simulation problems?
2. Can our functional solution be executed efficiently?

We discuss these individually in the next sections.

7.1 Other Types of Simulation Problems

What aspects of our problem make it well-suited to a functional solution? First, it is a queueing problem, and queues are very naturally modeled with streams. We have not explored modeling non-queueing problems in functional languages, but we hope that by drawing on the full expressive power of Haskell, we will be able to develop elegant functional solutions for these problems as well. We will report on this as we explore it further.

Even as a queueing problem, our example is relatively simple in that it uses only FCFS service disciplines. This allows us to use streams of times without having to retain any other information about each job. In general, however, we might want to use a different service discipline. For example, our CPUs might use a FCFS-preemptive resume queue, where each of the job sources is assigned a priority, and an arriving higher-priority job preempts a running lower-priority job. This type of service discipline is easy to model functionally, but requires that we maintain the source of a job along with the times at which it enters and exits various queues. Of course, this same information must be maintained in a non-functional solution as well.

Our problem is also simple in that the mappings represented by the queues are such that each element of a new queue is a simple function of a small number of inputs. There are a number of ways in which the definition of a queue might be complicated. For example, an element of a queue might depend on previous elements of that same queue, or of its inputs. Consider modeling a binary AND gate in a logic circuit with output `OUT` and inputs `IN1` and `IN2`. We would like the queues to contain times corresponding to signal changes, but this means that when the signal changes in one of the input queues, the output may or may not change. (Consider the case where both inputs are 0, then one changes to 1). We are essentially relying on the current state of the gate to determine its next state. We can do this functionally by maintaining the state information (in this case, the last values of `IN1`, `IN2`, and `OUT`) in the queue itself, and having each new element of the queue rely on its previous elements.

We have only outlined solutions to these issues, and we expect to pursue them more rigorously in future research.

7.2 Efficiency Issues

Recent work has produced a number of efficient implementations for lazy functional languages, but the execution time required for a particular application is hard to predict. Unfortunately, only a preliminary version of Haskell is currently available, so this point cannot be fully addressed at this time. However, based on past experience with functional languages, some issues can be anticipated. For example, it is well-known that lazy evaluation carries with it some overhead in the creation and evaluation of delayed objects. Some of this overhead can be eliminated through *strictness analysis* [Mycroft 1981], a compile-time analysis that eliminates lazy evaluation when doing so cannot change the semantics of the program. However, some laziness is necessary when streams are used, and we know of no data on how well strictness analysis performs on the sort of program described here. Other potential sources of inefficiency include list comprehensions and updatable structures, but much recent research has been directed at optimizing these constructs.

Overall, a functional solution to a large simulation problem may run slower than a solution in a traditional imperative language. However, the gain in readability, writability, and maintainability is considerable, and will likely outweigh any loss in execution efficiency. Furthermore, because functional programs have no notion of step-by-step execution, they contain implicit parallelism. For example, the solution described in this paper could be executed on a sequential or parallel machine without modifying the program. In the parallel case, a smart compiler such as that described in [Goldberg 1988] could automatically decompose the program into parallel units with appropriate granularities. Alternatively, some functional languages permit the programmer to annotate the program to assist the compiler in generating parallel code. The parallel execution of functional simulation programs is yet another area for future exploration.

ACKNOWLEDGEMENTS

Many thanks to Osman Balci for inspiring this work, and to Osman Balci and Marc Abrams for their helpful comments.

BIBLIOGRAPHY

Abrams, M. (1989), "A Common Programming Structure for Bryant-Chandy-Misra, Time-Warp, and Sequential Simulators," In *Proceedings of the 1989 Winter Simulation Conference,* E.A. MacNair, K.J. Musselman, and P.Heidelberger, Eds. IEEE, Piscataway, NJ, 661-670.

Bagrodia, R.L. and W. Liao (1990), "Maisie: A Language and Optimizing Environment for Distributed Simulation," In *Distributed Simulation,* Society for Computing and Simulation, San Diego, CA, 205-210.

Balci, O. (1988), "The Implementation of Four Conceptual Frameworks for Simulation Modeling in High-level Languages," In *Proceedings of the 1988 Winter Simulation Conference,* M.A. Abrams, P.L. Haigh, and J.C. Comfort, Eds. IEEE, Piscataway, NJ, 287-295.

Bryan, O.F. Jr. (1989), "ModSimII: An Object Oriented Simulation Language for Sequential and Parallel Processors," In *Proceedings of the 1989 Winter Simulation Conference,* E.A. MacNair, K.J. Musselman, and P.Heidelberger, Eds. IEEE, Piscataway, NJ, 172-179.

Hudak, P. and S. Anderson (1988), "Haskell Solutions to the Language Session Problems at the 1988 Salishan High-Speed Computing Conference," Research Report YALEU/DCS/RR-627, Department of Computer Science, Yale University, New Haven, CT.

Hudak, P. and P. Wadler et al.(1988), "Report on the Programming Language Haskell," Research Report YALEU/DCS/RR-777, Department of Computer Science, Yale University, New Haven, CT.

Lomow, B. and D. Baezner (1989), "A Tutorial Introduction to Object-Oriented Simulation and Sim++," in *Proceedings if the 1989 Winter Simulation Conference,* E.A. MacNair, K.J. Musselman, and P.Heidelberger, Eds. IEEE, Piscataway, NJ, 140-146.

Mycroft, A. (1981), "Abstract Interpretation and Optimizing Transformations for Applicative Programs," Ph.D. Thesis, University of Edinburgh, Department of Computing, Edinburgh, Scotland.