Proceedings of the 1993 Winter Simulation Conference
G.W. Evans, M. Mollaghasemi, E.C. Russell, W.E. Biles (eds.)

TOOLS FOR FUNCTIONAL SIMULATION

Adrienne Bloss
Michael Keenan
Kimberly Johnson

Department of Computer Science
Roanoke College
Salem, Virginia 24153, U.S.A.

ABSTRACT

Functional languages are useful for some types of simulation programming because they offer a clean, high-level programming style with powerful features such as higher-order functions and lazy evaluation. However, since functional languages are not specialized for simulation, they lack the powerful tools offered in simulation languages. In this paper we propose a set of tools to support queueing simulation in the functional language Haskell. The tools are applied to three examples. **Key words and phrases:** Modeling methodology, functional programming, queueing problems.

1 INTRODUCTION

Simulation models are usually programmed in traditional imperative languages such as C, Pascal, and Fortran, or in special-purpose simulation languages such as GPSS, SLAM, SIMAN, and SIMSCRIPT. Traditional languages are not particularly well suited to simulation programming; they are based on the step-by-step computational model of the von Neumann architecture, and produce low-level and rather tedious simulation programs. Simulation languages produce high-level code, but are not always available or appropriate for a given problem. In a previous work (Bloss 1990), the first author speculated that functional languages such as LML, Miranda, and Haskell may be well suited to simulation programming, and outlined a functional approach to simulation. In this paper we use the functional language Haskell to develop high-level tools based on streams and stream transformers for the simulation of queueing systems. We apply these tools to three sample problems, and discuss the results with respect to elegance and efficiency.

The testbed for this work is the programming language Haskell. Haskell is a lazy, purely functional language that captures most of the major features of modern functional languages. Haskell is still under development; the programs in this paper were written in version 1.2 as described in the most recent report (Hudak et al. 1992), and compiled using version 0.999.4 of **hbc**, the Chalmers Haskell compiler.

At the writing of Bloss (1990), no working Haskell system was available. Extensions to that work and a rudimentary implementation appeared later (Engel 1991), as did comparisons of high-level and low-level functional approaches (Bloss and Keenan 1992). At the latter time, we concluded that the high-level approach was more elegant and at least as efficient as the low-level approach, which led us to generalize the high-level approach through simulation tools.

The next section provides a reader's introduction to Haskell. Section 3 presents the tools we have developed for modeling queueing problems in Haskell, and Section 4 shows how these tools can be applied to three sample problems. Section 5 discusses the issues of efficiency and elegance, and Section 6 presents directions for future work.

2 AN OVERVIEW OF HASKELL

A Haskell program is a collection of mutually recursive functions, and the meaning of a program is the value of its 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. Anonymous functions may be written using lambda notation, e.g., `\ x -> x * x` is the function that returns the square of its argument. Unlike in LISP, parentheses are used only for grouping, and are often used redundantly for clarity. Consider the following definition of the factorial function:

```
factorial :: Int -> Int -> Int
factorial n acc = if (n<=1)
                  then acc
```

```
        else factorial (n-1) (n*acc)
```

The first line gives the type of `factorial`: it takes two integers and returns an integer. (All functions in Haskell are *curried*, that is, they take their arguments one at a time.) Haskell is statically typed, and although type declarations are not required (the compiler can infer them), we will generally include them.

Many interesting and important features of Haskell are not discussed in this brief introduction. In particular, modules, polymorphism and type classes greatly influence the programming style encouraged in Haskell, but are omitted here. The interested reader should refer to the Haskell Report (Hudak et al. 1992) for more information. In the current discussion, we will concentrate on imparting a reading knowledge of the Haskell features that are important in later sections, 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 the rest of the paper.

2.1 Lists

The list is 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 `a -> [a] -> [a]` for any data type `a`, where `[a]` signifies the type "list of a". Function `head` returns the first element of a list, and function `tail` returns the list containing everything but the first element. Thus for an element `e` of type `a` and a list `l` of type `[a]`, `head(e:l) = e` and `tail(e:l) = l`.

2.2 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 :: (a -> b) -> [a] -> [b]
map f l = if l==[ ]
          then [ ]
          else (f (head l)):
               (map f (tail 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 `head l` and `b` is bound to `tail l`. (An identifier representing the list can be specified with the symbol `@`; e.g., `l@(x:xs)` is the list `l` whose head is `x` and whose tail is `xs`.) Note that `map` is polymorphic: for any types `a` and `b`, it takes a function of type `a-> b` and a list of elements of type `a` and returns a list of elements of type `b`.

When a function has multiple definitions because of 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 $-$. Thus the parentheses in both definitions of `map` are redundant, but the parentheses in `factorial` are necessary.

2.3 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, or *streams*, can be defined and manipulated. Consider the examples below:

```
ones :: [Int]
ones = 1 : ones
    --- the infinite list of ones

nums_from :: Int -> [Int]
nums_from n = n : (nums_from (n+1))
     -- the infinite list of integers
     -- starting at n
```

A stream can be manipulated like any list — it can be passed to or returned from a function, 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 programs in which 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 take in tensquares ensures that only a ten-element list will be produced:

```
nums::Int
nums = nums_from 1

take :: Int -> [Int] -> [Int]
take n []       = []
take 0 l        = []
take n (x:xs)   = x : (take (n-1) xs)

square :: Int -> Int
square x = x * x

tensquares :: [Int]
tensquares = take 10 (map square nums)
```

3 SIMULATION TOOLS IN HASKELL

What tools are useful in writing queueing simulations? We need some way to model queues and servers, and, of course, random numbers. We also need some way to collect statistics. We will discuss each of these components individually in the next sections.

3.1 Queues

Queues are modeled naturally by Haskell's infinite lists. Input queues and output queues often have different structures, so we have defined them separately. For our examples, we will assume that an input queue is a list of job arrival times, and an output queue is a list of time pairs representing job service times and job departure times. The input queue for one server may be derived from the output queue from other servers, and the output queue from one server may be divided to create input queues for a number of other servers. Thus our basic tools must include functions to merge and divide queues. The functions to merge input queues are shown below:

```
mergeQs :: InQ -> InQ -> InQ
mergeQs [] q = q
mergeQs q [] = q
mergeQs q1@(t1:rest1) q2@(t2:rest2)
     | t1 < t2   = t1 : mergeQs rest1 q2
     | otherwise = t2 : mergeQs q1 rest2

mergeall :: [InQ] -> InQ
mergeall [q] = q
mergeall (q:qs) = merge q (mergeall qs)
```

In mergeQs, we assume that each queue element is a time, and that we want to produce a time-ordered merge of the queues. This could be trivially extended to operate on queues of arbitrary structure, assuming that some fixed place in the structure contains the time used for the merge. Function mergeall simply uses mergeQs to merge all of the queues from a list into a single time-ordered queue.

The dividing function, divideQ, takes an output queue and returns a new input queue containing a specified percentage of the jobs from the original queue. The code is shown below:

```
divideQ :: OutQ -> Float -> Float ->
           [Float] -> InQ
divideQ ((t1,t2):ts) bot top (r:rs)
     | (r >= bot) && (r < top) =
               t2 : divideQ ts bot top rs
     | otherwise = divideQ ts bot top rs
```

The first parameter is the queue to be divided. Since dividing is most often done with output queues, we assume that this is a queue of pairs. (It would be trivial to define divideInQ and divideOutQ if needed.) Parameters bot and top define the percentage of jobs that will be taken from the queue, and r:rs represents the list of random numbers that determines whether the current job is taken. For example, suppose q1 is to be divided into q2 and q3, with 20% of the jobs going to q2 and 80% of the jobs going to q3. Then q2 would be defined by calling divideQ with parameters q1, 0.0, 0.2, and a list of 0/1 random numbers, and q3 would be defined by calling divideQ with parameters q1, 0.2, 0.8, and the *same* list of 0/1 random numbers.

3.2 Servers

Servers take input queues and produce output queues, and thus are naturally modelled by functions of type InQ -> OutQ. The user could write a function to represent each server, but since most servers do fundamentally the same thing, this seems redundant. We can use Haskell's higher-order functions to define a *server creating* function makeserver, which takes a random number seed, the mean service time (we assume exponential distributions, but this would be trivial to generalize), and a sequence of externally-determined time intervals during which the server is available. These intervals represent finite "windows" during which jobs may be served. For example, an automatic teller machine might have scheduled down time several times a day for balancing; customers who arrive during a down time must wait until the machine comes back up, that is, until a "service window" begins.

Given these parameters, function makeserver returns a server that takes an input queue and returns an output queue, servicing each job according to the mean and windows indicated. A Haskell definition for makeserver is given in Figure 1. As a tool it should require little modification for most applications, but it is comforting nonetheless to see that it is relatively short and straightforward.

The real work here is done by really_server. It takes a list of random numbers, the next time at which the server is available, an input queue, and a list of time intervals during which the server is available from external forces. If the arrival time of the next job falls during an interval, the job is processed and put on the output queue; otherwise, really_server is called recursively with the appropriate parameters to let the job be served during the next window.

3.3 Random Numbers

Most discrete-event simulations are based on random numbers. In imperative languages it is easy to write a function rand with a modifiable seed parameter that returns a new pseudo-random number each time it is called. Such a function relies on side-effects, and therefore cannot be written in a functional language. However, random numbers may be modelled functionally using streams. First, we define a stream of uniformly distributed pseudo-random numbers. We use a multiplicative congruential random number generator, in which the i^{th} random number depends on the $(i-1)^{st}$ number recursively, i.e., $r_i = a * r_{i-1}$ *mod* m for some values of a and m. Given an initial integer value seed, a stream of random numbers between 0 and 1 may be defined as follows:

```
rand :: Int -> [Float]
rand seed = ((fromIntegral seed)/m) :
            (rand ((a * seed) 'mod' m))
```

(The fromIntegral is needed to prepare the integer value seed for real division.) Thus rand seed= [seed/m, (a*seed mod m)/m, (a*(a*seed mod m) mod m)/m, ...

Given this stream of 0/1 random numbers, a stream of exponentially distributed numbers with mean mu is easily defined by converting each number in the usual way:

```
expon :: [Float] -> Float -> [Float]
expon randlist mu =
       map (\r -> -(log r) / mu) randlist
```

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

```
cumul_expon :: Float -> [Float]
cumul_expon mu  = accumulate 0
                    (expon randlist mu)

accumulate :: Float -> [Float] -> [Float]
accumulate x l =
        let y = x + (head l)
        in  y : (accumulate y (tail l))
```

Clearly, other probability distributions could be modeled as well.

3.4 Statistic Collection

Statistic collection is straightforward in traditional simulation: when something of interest happens, the event is recorded in the appropriate global variables. In functional simulation this approach is not possible, as modifiable global variables cannot exist. Instead, since statistics are the desired output of a simulation, the statistics collector must *drive* the simulation. Recall that under lazy evaluation, an expression is evaluated only when its value is demanded in computing the final result of the program. Thus in our simulations, queues will be generated only to the extent that their elements are required for statistics collection. Of course, many different kinds of statistics may be collected, so it is difficult to make the collection fully general. However, the changes from one sort of statistic to another are slight. Most statistics are derived from the servers' output queues; if more information is carried along with each job, more statistics can be computed. Consider the definition of compute_stats below (the return type Dialogue simply indicates that compute_stats prints its result):

```
compute_stats :: [OutQ] -> [Stat] ->
                  Int -> Dialogue
compute_stats qs s n
 | allEmpty qs = appendChan stdout
                 (shows (n,
                 (map (\stat -> stat)) s))
                      "\n")
                 abort  done
 | otherwise =
        let heads = map maybe_head qs
        in   compute_stats
             (map maybe_tail qs)
             (upd_stats heads oldstats)
             (n + 1)
```

This function is very general in that it simply processes each job from each queue it is given according to upd_stats, which determines exactly which stats are computed. Upd_stats is straightforward; it could be written specifically for a given simulation problem,

```
makeserver :: Float -> Int -> [(Time,Time)] -> InQ -> OutQ
makeserver mu seed patt  =
    let really_server times@(t:ts) avail jobs@(j:js) windows@((b,e):rest)
                | avail < b  = really_server times b jobs windows
                | j < e_time = let served = max j avail
                                  done   = served + t
                              in  (served, done) :  really_server ts done js windows
                | otherwise  = really_server times avail jobs rest
    in \q -> really_server (expon (rand01 seed) mu) 0 q windows
```

Figure 1: Haskell code for makeserver

or a general version could offer a variety of possible statistics, and the programmer would simply select those that were desired. For example, to compute the utilization for each server, we use upd_stats to sum the durations of that server's service times:

```
upd_stats :: Time -> Time -> OutQ ->
                    [Stat] -> [Stat]
upd_stats begin end [] [] = []
upd_stats begin end ((t1,t2):t2) (s:ss) =
  (s + ((min t2 end) - (max begin t1))) :
  (upd_stats t2 ss)
```

A slight modification would also be required in compute_stats so that it divided each statistic by total system time before printing.

4 APPLICATIONS

The tools described in the previous section can be assessed by applying them to real simulation problems. In this section we present a detailed solution to one problem using these tools, and outline their use in the solutions to two additional problems.

4.1 Computer System Simulation

4.1.1 Description of Problem

The first problem is taken from Balci (1988), and is the same problem that was used in two previous works (Bloss 1990, Bloss and Keenan 1992). 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 batch computer system operates with two central processing units (CPUs). Jobs submitted to the system come from four sources: (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 connected to the local area network (LAN) with 9600 baud rate. Assume that the interarrival times of batch programs to the system with respect to each user type are determined to have an exponential probability distribution with means of 3200, 640, 1600, and 266.67 for the 300, 1200, 2400, and 9600 baud users respectively.

A submitted batch program first goes to the job entry subsystem (JES). 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 with a probability of 0.2 or to the printer (PRT) with a probability of 0.8. All queues are handled by a first-come-first-served discipline, and 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.

The structure of the system is shown in Figure 2.

4.1.2 Solution in Haskell

A Haskell solution to the problem described above is shown in Figure 3. The servers are represented by functions `jess`, `cpu1`, `cpu2` and `printer`, and are defined by passing the appropriate mean service times and service windows to `makeserver`. Note that for this problem, the windows encompass the entire simulation time, since there are no outside restrictions on server availability. Thus `start_end_list` is simply a list containing the single pair of the starting and ending times of the simulation.

The queues are defined in a straightforward way in terms of the initial queues, the servers, and the appropriate divides and merges. These definitions should be self-explanatory: `q1` is the merge of the four initial queues, each of which is a cumulative exponential dis-

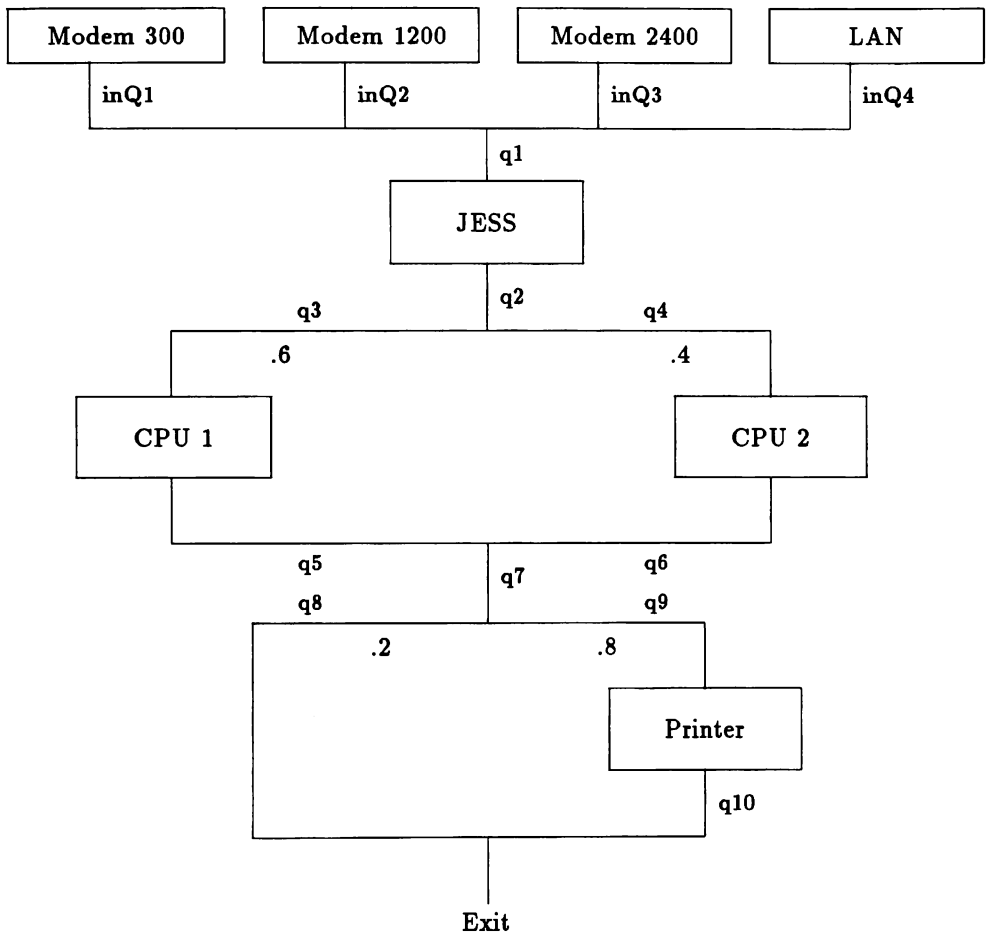

Figure 2: Computer system; queue labels refer to Haskell code

```
stats begin end =
    let
        q1 = let inQ1 = cumul_expon (rand seed1) 3200.0
                 inQ2 = cumul_expon (rand seed2) 640.0
                 inQ3 = cumul_expon (rand seed3) 1600.0
                 inQ4 = cumul_expon (rand seed4) 266.67
             in mergeall [inQ1, inQ2, inQ3, inQ4]
        q2 = jess q1
        q3 = divide q2 0.0 0.6 divideRands
        q4 = divide q2 0.6 1.0 divideRands
        q5 = cpu1 q3
        q6 = cpu2 q4
        q7 = mergeOutQs q5 q6
        q8 = divide q7 0.0 0.2 divideRands2
        q9 = divide q7 0.2 1.0 divideRands2
        q10 = printer q9
        jess = makeserver 112.0 seed5 start_end_list
        cpu1 = makeserver 226.67 seed6 start_end_list
        cpu2 = makeserver 300.0 seed7 start_end_list
        printer = makeserver 160.0 seed8 start_end_list
        divideRands = rand seed
        divideRands2 = rand seed9
        start_end_list = [(0,end)]
```

Figure 3: Haskell code corresponding to labelled system in Figure 2.

tribution with a given mean; q2 is the output queue produced from q1 by the JESS; q3 and q4 are derived from q2, with 60% of the jobs in q2 going to q3 and 40% going to q4; and so on. These queues are labeled in Figure 2.

The only pieces missing from the code shown in Figure 2 are the tools described in Section 3, the definitions for the random number seeds (constants), and a `main` function that reads in a start and end time and calls the statistics generator. Thus Figure 3 truly represents the entire substantive program; the remarkable thing is how closely it resembles the diagram in Figure 2.

4.2 Other Applications

After writing the computer system simulation described above, we used the tools described in Section 3 to write two additional simulations. Although space constraints prohibit detailed descriptions here, these problems and their solutions are outlined in this section.

The first problem is a simple intersection controlled by a single traffic light. Both streets carry one-way traffic, and no turns are permitted. Given the cycle time of the light and the distributions of car arrivals, we wish to determine the average time for which cars traveling in each direction must wait at the intersection.

The second problem is a recycling center with an entrance station, five recycling workstations, and an exit station. Each workstation handles a single type of material — aluminum, plastic, etc. — and each truck carries a single type of material. There are several types of trucks, with different capacities and different probabilities of carrying each material. Each truck stops at the entrance station, proceeds to the appropriate workstation, waits to be serviced there, then proceeds to the exit station. The center has three workers that float among the five workstations; a truck cannot unload at a workstation until it is assisted by a worker, and only one truck can unload at a given workstation at a time. Within a workstation, trucks are served on a first-come-first-serve basis. We wish to determine average wait time at each server, and average total time in the system.

For both of these problems, the tools in Section 3 were helpful in designing Haskell solutions. For the intersection problem, the tools applied directly and the resulting program is very simple. Only about 10 lines of code are substantially different from the computer system solution, and only 10 or so additional

lines are even trivially different.

The recycling center program is more complex, and the tools in Section 3 were not general enough to do all the work. Nevertheless, they served as templates from which the appropriate code was easily derived. For example, the three kinds of servers (entrance, workstation, and exit) are not generated directly by the `makeserver` function, but in each case only modest modifications were necessary. Furthermore, the top-level code clearly reflects the structure of the problem.

5 COMPARING HASKELL TO C AND SIMSCRIPT

For each of the three problems above, we looked at the elegance, code size, and runtime efficiency of the solutions coded in Haskell, SIMSCRIPT, and C. (We did not construct a C solution for the recycling center.) Our observations are described below, and the length and runtime figures are shown in Tables 1 and 2.

Elegance is the hallmark of functional languages, and in many ways this elegance is apparent in our Haskell solutions. The Haskell programs are modular and well-structured, and we were able to make use of the simulation tools described in Section 3. On the other hand, the lack of a global modifiable state proved clumsy; we were forced to carry around a lot of information that should have been local to one or two functions. Nevertheless, in every case the top-level code clearly reflects the structure of the problem, and the underlying code is fairly straightforward. This is in contrast to the SIMSCRIPT and C solutions, both of which spread the structure of the simulation throughout the code.

Counting lines of code provides only a rough measure of program size, but we felt that it was important to get some idea of the relative bulk of the simulations in the three languages. Because of the powerful simulation tools provided in SIMSCRIPT, we expected the SIMSCRIPT programs to be significantly shorter than the Haskell or C programs. But as shown in Table 1, the Haskell programs are slightly shorter than the SIMSCRIPT programs, with the C programs predictably much longer than either. It should be noted, however, that the SIMSCRIPT programs contain about four times as many lines of "trivial" code (constant definitions, prints statements, etc.) as the Haskell programs, and the SIMSCRIPT lines are, on average, somewhat shorter than the Haskell lines. But even accounting for these factors, the Haskell programs are reasonably concise.

The runtime figures in Table 2 must be viewed as approximate, because we were unable to run all the programs on the same platform. The SIMSCRIPT programs were run on a VAX 4000; the C and Haskell programs were run on a DECstation 3100. In our configurations, the DECstation seems to be about twice as fast as the VAX, so the SIMSCRIPT times have been divided by two to produce comparable figures. Thus for the computer system and the traffic light, the Haskell times are close to the SIMSCRIPT times, but the recycling center time is slower by a factor of 500. Why is there such a big discrepancy? We suspect that the Haskell recycling center code contains a "space leak," that is, that some interaction between the programming style and the compiler optimizations cause it to use much more memory than necessary. We are investigating this and other possibilities, and hope to report on them soon.

PROBLEM	HASKELL	SIMSCRIPT	C
Computer System	91	109	205
Traffic Light	69	90	347
Recycling Center	249	275	-

Table 1: Lines of code in sample simulations

PROBLEM	HASKELL	SIMSCRIPT	C
Computer System (400,000 sec)	3.0	3.0	0.5
Traffic Light (149,000 sec)	13.0	3.4	5.0
Recycling Center (10,000 trucks)	580	12.5	-

Table 2: Runtimes in sample simulations (seconds)

6 CONCLUSIONS AND FUTURE WORK

We found that a small set of tools could be very useful in coding queueing simulations elegantly and reasonably efficiently in Haskell, and we are very encouraged by this result. We have identified several directions for future work:

1. Identify and eliminate the cause of the poor execution time for the Haskell recycling center pro-

gram.

2. Investigate ways to reduce the clutter caused by the lack of a global state. In particular, determine whether some form of *functional state* based on monads (Peyton Jones and Wadler 1993, Wadler 1992) or linear type systems (Girard 1987, Guzman and Hudak 1990, Wadler 1990) can help here.

3. Investigate the implications of these tools for parallel evaluation; is sufficient parallelism available, and if not, can the amount of parallelism be increased? What about parallelism in a state-based approach?

4. Investigate the application of Haskell to non-queueing problems in discrete event simulation.

REFERENCES

Baoci, O. 1988. The implementation of four conceptual frameworks for simulation modeling in high-level languages. In *Proceedings of the 1988 Winter Simulation Conference,* ed. M.A. Abrams, P.L. Haigh, and J.C. Comfort. IEEE, Piscataway, NJ, 287-295.

Bloss, A. 1990. A functional approach to simulation. In *Proceedings of the 1990 Winter Simulation Conference,* ed. O. Balci, R.P.. Sadowski, and R.E. Nance. IEEE, Piscataway, NJ, 214-219.

Bloss, A. Keenan, M. 1992. Discrete event simulation in the functional language Haskell. In *Proceedings of the 22nd Annual Virginia Computer Users Conference,* Virginia Tech, 57-76.

Engel, J. 1991. Design of a discrete digital simulation in Haskell. Undergraduate honors thesis, Department of Computer Science, Virginia Tech, Blacksburg, Virginia.

Girard, J-Y. 1987. Linear logic. *Theoretical Computer Science,* 50:1-102.

Guzman, J. Hudak, P. 1990. Single-threaded polymorphic lambda calculus. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science.*

Hudak, P. Wadler, P. et al. 1992. Report on the programming language Haskell, version 1.2. *ACM SIGPLAN Notices,* 27(5).

Peyton Jones, S.L. Wadler, P. 1993. Imperative functional programming. In *Proceedings of the 20th Annual Symposium on Principles of Programming Languages,* 71-84, ACM Press.

Wadler, P. 1990. Linear types can change the world. In *Programming Concepts and Methods,* ed. M. Broy and C.B. Jones. North Holland.

Wadler, P. 1992. The essence of functional programming. In *Proceedings of the 19th Annual Symposium on Principles of Programming Languages,* ACM Press.

AUTHOR BIOGRAPHIES

ADRIENNE BLOSS is an Assistant Professor of Computer Science at Roanoke College. Her current research interests focus on real-world applications for functional languages.

MICHAEL KEENAN is a Visiting Associate Professor of Computer Science at Virginia Tech. His research interests include functional programming and complexity theory.

KIMBERLY JOHNSON is a senior at Roanoke College, majoring in Computer Science and Computer Information Systems. She joined this project in January 1993, and stayed on through a grant from the College.