A General Framework for Discrete-event Simulation Using Functional Languages

A.J. Field and R.L. While

Abstract

In this paper we explore the use of functional programming languages in the design and implementation of discrete-event simulation programs. We show, in particular, how language features such as polmorphism, higher-order functions and lazy evaluation can be used to develop powerful, re-usable, library modules for, among others, event management, simulation monitoring and statistical analysis. We give examples of simulation programs developed within the proposed framework; these illustrate the conciseness of the functional notation and at the same time demonstrate how the expressive power of functional languages can be used in applications of this nature.

1 Introduction

In this paper we address the software aspects of simulation and focus on the rôle of modern functional language technology in the design and implementation of discrete-event simulation codes. We are particularly concerned, therefore, with programmer productivity and the rôle functional languages can play in the development of concise, easily written codes that are both reliable and easily understood.

2 A Functional Language

The notation we shall use in this paper is an extension to that of the Miranda programming language, as described in [1]. A program script consists of type definitions and a collection of (possibly mutually recursive) function definitions. The important features of the language are summarised as follows using examples.

Polymorphism— Each function in a program has an associated *type*. These types may be *polymorphic*, i.e. parameterised by objects of generic type. We make extensive use of the built-in polymorphic list type `[*]`, read as "list of anything". An example is the 'length' function on lists:-

```
length :: [*] -> num
length []         = 0
length ( x : xs ) = 1 + length xs
```

The empty list is written `[]` and a non-empty list with head element `x` and tail `xs` is written `x:xs` where '`:`' is the (infix) list constructor. The first line of the example declares explicitly the type of the `length` function; the symbol `*` denotes a *type variable* which can be instantiated to any type, as in `[num]` - which would be the type "list of *numbers*". Note that argument structures are decomposed by *pattern matching*.

Higher-order Functions— A higher-order function is a function which takes another function as an argument and/or which delivers a function as its result. An example is the function `map` which applies a given function to each element of a given list:-

```
map :: ( * -> ** ) -> [*] -> [**]
map f []         = []
map f ( x : xs ) = f x : map f xs
```

Another function to which we shall refer later on is a polymorphic 'min' function which delivers the 'smallest' element in a list of arbitrary object types, the object oredering ordering function is supplied as a parameter:-

```
min :: ( * -> * -> bool ) -> [*] -> *
min less ( x : [] ) = x
min less ( x : xs ) = x, if less x m
                    = m, otherwise
                      where
                        m = min less xs
```

Objects of function type are created by the *partial application* of an existing function as in

```
allmins :: [[num]] -> [num]
allmins = map ( min (<) )
```

The `min` function is shown partially applied to the function '<' (the enclosing brackets converts '<' from infix to prefix function form). The result of the partial application is the standard ordering function *on numbers*.

Record Types— To simplify our notation we shall allow *record* types, which are simply tagged versions of conventional product (tuple) types—these are not a feature of Miranda but are easily mimicked at some notational expense. The general format for a record type will be as follows:-

```
T == ( f1 :: T1, ..., fn :: Tn )
```

where, in keeping with the functional style, `f1, ..., fn` are *projectors* and `T, T1, ..., Tn` are types (these may be parameterised by one or more type variables). Projector `fk` is a function of type `T -> Tk`, $1 \leq$`k` $\leq$`n`. Thus, if `r = (v1,...,vn) :: T` then `fk r = vk`, $1 \leq$ `k` $\leq$ `n`.

We shall further extend the syntax to allow record modification to be defined succinctly. We will assume that in addition to the implicit introduction of projectors, the record type declaration also introduces implicitly a corresponding set of update functions. If `f` is an implicitly defined projector of type `T' -> T''`, then `*f` is an implicitly defined update function of type `( T'' -> T'' ) -> T' -> T'`, defined axiomatically by

```
f ( *f g x ) = g ( f x )
```

Thus, using the previous example, the expression `*fk g r` denotes the record `(v1, ..., g vk,..., vn)`, $1 \leq$`k` $\leq$`n`.

Observe now that multiple record updates may be specified by the *composition* of unit updates. For example, with `r` defined as above, `(*fj g` $\circ$ `*fk h) r`, $1 \leq$`j,k` $\leq$`n, j`$\neq$`k` defines the new record `r'` satisfying `fj r' = g vj, fk r' = h vk, fi r' = vi, i`$\neq$`j, i`$\neq$`k`. The composition operator $\circ$ has the usual definition `( f` $\circ$ `g ) x = g ( f x )`.

Infinite Structures— In keeping with most functional languages we shall assume the evaluator to be *lazy*, meaning that the arguments to functions are not evaluated until their values are required in the body. In particular this allows the construction and exploration of infinite lists of values since neither the head nor tail arguments of ':' are evaluated until needed. We will use this facility to model streams of simulation input. For example, an infinite sequence of $U(0,1)$ samples may be constructed by the multiplicative congruential method as follows:-

```
a = 16387
m = 1073741823
rand :: num -> [num]
rand seed = next : rand newseed
            where
              newseed = ( seed * a ) mod m
              next = newseed / m
```

where `seed` is any positive odd integer.

These values can be mapped to samples from other distributions. As an example the standard polar coordinate transform method for generating standard normal samples, and the transform method for generating exponenentially-distributed samples, can be defined as follows:-

```
norm ( r : r' : rs )
  = x * cos y : x * sin y : norm rs
    where
      x  = -2 * ln r
      y  =  2 * pi * r'

exp lam ( r : rs )
  = - ln r / lam : exp rs
```

One or more samples can be 'picked off' from such infinite sequences either by pattern matching, or by the application of the list projectors `hd` and `tl` defined by `hd (x:xs) = x, tl (x:xs) = xs`.

Queues— Avoiding efficiency arguments at this stage we shall implement queues using lists. We shall use ploymorphism to enable the same queue data type to be re-used. To allow different queueing disciplines the `enqueue` function is parameterised by the queue ordering function.

```
queue * == [*]

emptyq :: queue *
emptyq = []

isemptyq :: queue * -> bool
isemptyq []  = True
isemptyq any = False

qlength = length

enqueue :: ( * -> * -> bool ) -> * -> queue * -> queue *
enqueue p x []          = [x]
enqueue p x ( y : ys )  = x : y : ys,          if p x y
                        = y : enqueue PRI x ys, otherwise

dequeue :: queue * -> ( *, queue * )
dequeue ( x : xs ) = ( x, xs )
```

Note that by varying the ordering function supplied to `enqueue` we can specify, for example, Last-In-First-Out (`LIFO x y = True`), First-In-First-Out (`FIFO x y = False`) or priority queueing (`PRI x y = True` iff `x` has higher priority than `y`).

3 Event Management

We shall implement the event 'diary' (the time-ordered sequence of pending event occurrences) as a queue of event/time pairs, ordered on the second (time) component, i.e. of type `queue ( event, num )` where time is represented as a number. Each event will be modelled as a *function* which will be placed *within* each diary entry. The event manager's rôle is then to repeatedly *call* the next event function in the diary, supplying it with both the time of the event occurrence and the current simulation state. Each event function will be defined to deliver a pair of update functions—one for the state and one for the diary. Before iterating, the manager must apply these functions to effect the required state and diary updates.

To keep the event manager generic, it is defined to accept a completion *function*, `comp`, a function of both the current time and state, as an additional parameter. The above process will be repeated until this function delivers `True`.

More generally, we may wish to trigger events on the basis of the current state or time. To effect this we supply two additional functions to the event manager, namely a trigger function, `trig`, with the same type as `comp`, which invokes the response event, `resp`[1], when the trigger

[1]The response event is effectively scheduled at the *current* time i.e. before the next event occurrence in the diary.

function returns `True`. We obtain:-

```
diary *   == queue ( event *, num )

compfun * == * -> num -> bool

trigfun * == * -> num -> bool

manage :: * -> diary * -> compfun * -> trigfun * -> event *
          -> ( *, num )
manage s d comp trig resp
  = ( s, t ),                                      if comp s t
  = manage ( ups' s ) ( upd' d ) comp trig resp, if trig s t
  = manage ( ups s  ) ( upd d' ) comp trig resp, otherwise
    where
      ( ups, upd )   = ev s t
      ( ups', upd' ) = resp s t
      ( ( ev, t ), d' ) = dequeue d

notrig s t = False

noresp s t = ( s, t )
```

Notice that the result of each run is a pair `(s,t)` where `t` is the completion time and `s` is the terminal state.

This now exposes the required type for each event function in a simulation:-

```
event * == * -> num -> ( * -> *, diary * -> diary * )
```

(Note: this type is circular since the diary itself contains instances of event functions. The type checker will need to spot this circularity—see [2] for details)

Event Scheduling—The scheduling of an event is now achieved by means of a sceduling *function*, defined in terms of `enqueue`:-

```
sched :: event * -> num -> diary * -> diary *
sched ev t = enqueue PRI ( ev, t )
             where
               PRI ( e', t' ) ( e'', t'' ) = t' < t''
```

To schedule the single event `e` at num `t` the *partial application* `sched e t` (a diary update function) is returned to `manage` as the second component of the event result pair. The scheduling of two or more event occurrences is then achieved by the *composition* of such partial applications, similar to the composition of state updates illustrated above.

Initialisation— Initialisation involves calling the function `manage` with an initial state and diary update function in addition to the completion criteria function and the `trigger` and `response` functions:-

```
startsim :: * -> ( diary * -> diary * ) -> compfun *
         -> trigfun * -> event * -> ( *, num )
startsim s upd = manage s ( upd emptyq )
```

The functions so far defined constitute a generic event management library which we assume to be imported by all simulation codes.

4 Example: A Multiple Server Queueing System

We have room to illustrate one example of the use of the generic functional event manager. We consider a system comprising a single, infinite-capacity, queue with `n`, `n`$\geq$1 servers. We assume

that customer arrivals are Poisson with rate `lam` and that the service times are exponentially distributed with rate `mu`. We shall monitor the customer waiting times to illustrate how measurement variables may be incorporated into the simulation state. The minimal state for the simulation includes the state variables (the queue `q` and the number of idle servers available `nserv`), the random input sequences (the infinite list of inter-arrival time samples `iats` and service time samples `sts`), the measurement variables (here the total observed waiting time `totw` and the total number of customers served `n`) and finally the simulation control variables; we assume here that the simulation executes for a fixed simulated `runtime`. Since we are only interested in customer waiting times, we need only record the arrival time of each customer.

```
state == ( q       :: queue num,   nserv :: num,
           iats    :: [num],       sts   :: [num],
           totw    :: num,         n     :: num,
           runtime :: num )
```

Since there may be several departure events in the diary, we distinguish them by associating with them the arrival time of the customer they being served. We can achieve this *without requiring a change to the event manager* using higher-order functions: we partially apply the departure function to customer arrival time, essentially coupling the two in the event diary. The arrival and departure functions are then as follows

```
arr s t = ( ups,  nextarr ∘ nextdep ), if nserv s > 0
        = ( ups', nextarr ),           otherwise
          where
            ups  = *nserv dec ∘ discard
            ups' = *q ( enqueue FIFO t ) ∘ discard
            discard = *iats tl ∘ *sts tl
            nextarr = sched arr ( t + hd ( iats s ) )
            nextdep = sched ( dep t ) ( t + hd ( sts s ) )

dep :: num -> event state
dep at s t = ( ups, nextdep ), if ~( isemptyq ( q s ) )
           = ( ups', id ),     otherwise
             where
               ups  = updatewt ∘ *q ( K backq ) ∘ *sts tl
               ups' = updatewt ∘ *nserv inc
               nextdep       = sched ( dep at' ) ( t + st )
               ( at', backq ) = dequeue ( q s )
               updatewt      = *totw ( (+) ( t-at ) ) ∘ *n inc
```

The result of the simulation run is the final state and the completion time, from which an output report is normally produced. The top-level function call will therefore have the following structure (the report formatting function, `report`, is omitted, although it is easily enough defined):-

```
run :: num -> num -> num -> num -> num -> ( num, num )
run lam mu runtime seed1 seed2
  = report ( startsim s ( sched arr 0 ) stop notrig noresp )
    where
      stop s t = ( t > runtime )
      s    = ( emptyq, True,
               exp lam ( rand seed1 ),
               exp mu ( rand seed2 ),
               0, 0, runtime )
```

where `notrig` and `noresp` are the null trigger and response functions defined earlier. This example does not require the triggering mechanism provided by the manager. An example of an application which does, is 'batched' simulation in which the run is divided into 'n' batches each of which observes the system for 'k' customer completions, to obtain a confidence interval, for example. Here the trigger will check for 'k' completions in one batch and the response event will reset the simulation statistics and update a result accumulator (in the state) in preparation for the next batch. Space constraints preclude a listing of the required code.

5 Summary and Conclusion

In this paper we have investigated how functional programming techniques can be usefully exploited in the development of discrete-event simulation programs. We have developed a generic event management library which uses higher-order functions to give the event diary the basic properties of an abstract data object, and which exploits polymorphic type checking to make the management functions re-usable across different codes.

The general framework described has been used extensively to teach undergraduate simualation courses. The functional notation leads to short, easily understood programs which are more concise than their procedural program equivalents whose more verbose syntax and less orthogonal semantics leads to codes which are often three or four times the size.

From the software engineering point of view, when compared to simulation-specific programming languages such as GASP [3] and SIMSCRIPT [5], our approach leads to similarly structured codes, but at a much higher level of abstraction. In particular the modular delineation of simulation-specific state and event procedure definitions, and the generic event management functions and supporting library tools is broadly the same in both cases. What is particularly significant, however, is that these features can be provided entirely within the functional language framework *by the user* rather than requiring special language "features" to be supported syntactically. With the full repertoire of functional language features available to the programmer, more general processing, e.g. of simulation state information or simulation output, is significantly easier and almost universally more concise than in a more conventional language.

References

[1] Bird, R. and Wadler, P., "Introduction to Functional Programming", Prentice Hall International, 1989.

[2] Cordone, F. and Coppo, M., "Recursive Types: Syntax and Semantics", In *Information and Computation*, Vol. 92, May 1991. November 1986.

[3] Pritsker, A.A.B, "The GASP IV Simulation Language", Wiley, New York, 1975.

[4] IBM Corporation, "General Purpose Simulation System V, User's Manual", Form SH 20-0851, White Plains, New York.

[5] Consolidated Analysis Centres Inc., "SIMSCRIPT II.5 Reference Handbook", Los Angeles, 1972.

Appendix: Some Primitive Functions Used

Primitives on Lists

`#`	The prefix 'length' function
`++`	The infix append function
`hd`	Delivers the head of a given list
`tl`	Delivers the tail of a given list
`!`	The infix list indexing function (note `xs ! 0 = hd xs`)
`take`	`take n xs` delivers the first `n` elements of `xs`
`drop`	Defined axiomatically by `take n xs ++ drop n xs = xs`
`postfix`	Defined by `postfix x xs = xs ++ [ x ]`
`sum`	Delivers the sum of the elements of a given list (of numbers)

Miscellaneous Functions

`K`	The 'K' combinator, defined by `K x y = x`
∘	The infix function composition operator