Lambdas in the Liftshaft – Functional Programming and an Embedded Architecture

Malcolm Wallace and Colin Runciman
Department of Computer Science
University of York, YO1 5DD, United Kingdom
{malcolm,colin}@minster.york.ac.uk

Abstract

Embedded computer systems seem to be the antithesis of functional language systems. Embedded systems are small, stand-alone, and are often forced to accept inelegant design compromises due to hardware cost. They run continuously and are reactive, that is, their primary goal is to monitor sensors and control effectors, using observed external events to trigger state-changing control actions. Yet this paper describes how functional abstraction can tame the inelegance of embedded systems. Architectural compromises can be made in device drivers, programmed within the functional language, but a function-level interface is presented to the application programmer. Examples are given from a liftshaft case study.

1 Introduction

Programming an embedded system requires certain features to be present in the chosen programming language and environment. It must be possible:

- to write detailed device-driving code;
- to deal promptly with asynchronous events (interrupts);
- to ensure that the system can run continuously and indefinitely.

As Modula extended Pascal for systems programming, so *Embedded Gofer* [Wallace95] extends Haskell for the same purpose. Embedded Gofer is a variant of Gofer [Jones94] permitting static processes, I/O register access, and interrupts. An incremental real-time garbage collector [WR93] is used to ensure that there are no pauses for memory management.

An earlier paper [WR95] introduced the main extensions to Gofer. This paper demonstrates, through a demanding example application, firstly that device driving can be performed in a general-purpose functional language, and secondly that suitable abstractions can hide these device details from the rest of the functional program. The functional implementation of the example application, a liftshaft controller, runs successfully on the apparatus described in §3.

Section 2 gives an overview of Embedded Gofer's facilities for systems programming. Section 3 describes the particulars of the liftshaft application. This example was chosen because the apparatus was originally built to teach embedded systems programming in Modula. The ''difficult'' aspects are highlighted in the following sections. Section 4 demonstrates a functional device driver which uses interrupts to determine the position of a liftcar. Section 5 describes some of the architectural compromises often encountered when programming sensors and effectors, and illustrates some functional solutions. Section 6 describes the controlling algorithm for the liftshaft, in which all interaction with devices is conveniently abstract. Section 7 reports our results. Section 8 compares Embedded Gofer with related work. Section 9 discusses future work. Section 10 concludes by summarising the significance of this work.

2 Overview of Embedded Gofer

Embedded Gofer extends Gofer in five ways: by the addition of processes, typechecked message-passing, I/O register access, a means of interrupt-handling, and real-time garbage collection.

2.1 Processes and typechecked message-passing

In embedded systems, device drivers should be modular and independent. To this end, Embedded Gofer provides communicating functional processes in the manner of Stoye's sorting office [Stoye86]. Processes are statically defined, and share information by passing messages to each other via the runtime system. Much routing of message traffic is statically typechecked [WR94] using Haskell's type classes and Gofer's constructor classes [Jones93].

Each process is a function performing monadic I/O [PJW93]. The two basic monadic operations for transmitting messages are `send` and `recv`.

```
class Process action msg where
  send :: RecipientFor x =>
          x -> action msg ()
  recv :: action msg msg
```

Every process may receive only one type of message, but may send messages of any type. Often, no address need be attached to a message, because the message's type determines its recipient. Where more than one process may receive messages of identical type, the message must contain some (programmer-defined) means to distinguish between potential recipients.

2.2 Register access

Device-driving processes need a means of reading and writing I/O registers. Two new monadic I/O primitives, *getReg* and *putReg*, are provided by Embedded Gofer. These do not allow access to general registers – only to registers in I/O space.

```
class DevProcess action msg where
  getReg :: Addr -> action msg Word
  putReg :: Addr -> Word
                  -> action msg ()
```

Processes which use device registers are of a more specialised type than processes which simply send and receive messages. Constructor classes are used to layer I/O *privileges* over the different process types.

2.3 Interrupts

A second layer of I/O privilege for device-handling processes is the ability to receive interrupts. Interrupts are treated as special messages in the input streams of particular processes. Such a process is statically associated with a unique interrupt vector. Handler processes are written such that their evaluation depends on the arrival of an interrupt. A pre-emptive run-time scheduler ensures that while handler processes are blocked awaiting interrupts, the evaluation of other processes is free to proceed. However the correct handler is evaluated immediately its interrupt occurs. This is possible because each process is independent – one process cannot side-effect another.

A handler process must select between interrupt input and ordinary message input. The new I/O primitive *select* is provided: an interrupt, if available, takes priority over a message, causing the appropriate branch to be returned. If no message or interrupt is available, the process blocks.

```
class IntProcess action msg where
  select ::
      (Interrupt -> action msg v) ->
            (msg -> action msg v) ->
                    action msg v
```

Embedded Gofer's functional interrupt-handling model is more fully described in [WR95].

2.4 Notation

We use the name *result* for the monadic combinator often called (confusingly) *unit* or *return*; the name *?* for *bind*; and the name *>>* for *bind_*. Some derived combinators are useful, especially *loopwith*, a higher-order name for tail-recursion with a local state.

```
loopwith s f =
    f s ? \t -> loopwith t f
```

2.5 Real-time garbage collection

Embedded Gofer has an incremental real-time garbage collection algorithm called *Stack-Safety* [WR93]. Based on incremental variants of Mark-Sweep, it guarantees a very small maximum time per increment of collection, and is designed to be used on machines with very small memories (and no virtual memory). Hence, embedded applications can be guaranteed to run continuously, i.e. without long pauses for GC.

3 The liftshaft

A model liftshaft (see Figure 1) stands approximately 1.5m high. There are two cars built of meccano, each of which has a corresponding counterweight. The two shafts are sectioned into six floors, with a platform for each floor. There is one call button marked ''Up'' on all floors except the highest, and one marked ''Down'' on all floors except the lowest. The call buttons are located between the two shafts, and each has an internal indicator lamp which can be lit under software control.

Each lift-car has six request buttons, one per floor. These buttons do not have internal lamps. Each car also has a seven-segment LED display and a loudspeaker capable of making a ''bleep'' noise.

As shown in Figure 2, for each car, a string of fixed length is attached to the roof of the construction in two places – above the car and above the counterweight. The car and the counterweight are suspended on this string by pulley wheels. The middle section of the string passes through a series of pulley wheels above the roof of the unit. One of the wheels is driven directly by a low-voltage bi-directional motor, with a maximum speed of 60 r.p.m.; another is connected to a precisely-milled telemetry wheel and optical detection system; the third wheel provides tensioning.

Four end-stop switches provide a fail-safe mechanism for the cars. When a car reaches the top or bottom of the shaft, it trips one of these switches which automatically cuts the power to the motor†.

The liftshaft is controlled using a single-board microcomputer based around the Motorola 68010 central

† This reduces wear and tear on suspending strings and motors.

processor. Two linked custom-built interface cards provide some extra control devices and also hold logic to translate control signals into motor voltages. Two Motorola 68230 Parallel Interface/Timer (PIT) chips are connected to the motor, lights, telemetry wheels, end-stops, and shaft buttons. One Motorola 68681 Dual Asynchronous Receiver/Transmitter (DUART) connects to a host computer via two serial lines. These lines are used to download programs and display program output. A second 68681 DUART connects to the liftcar buttons, seven-segment displays, and bleeps.

The hardware and controller boards were originally designed for teaching systems-programming in Modula. This complete embedded system is now programmed in a strongly-typed polymorphic functional language, without loss of referential transparency.

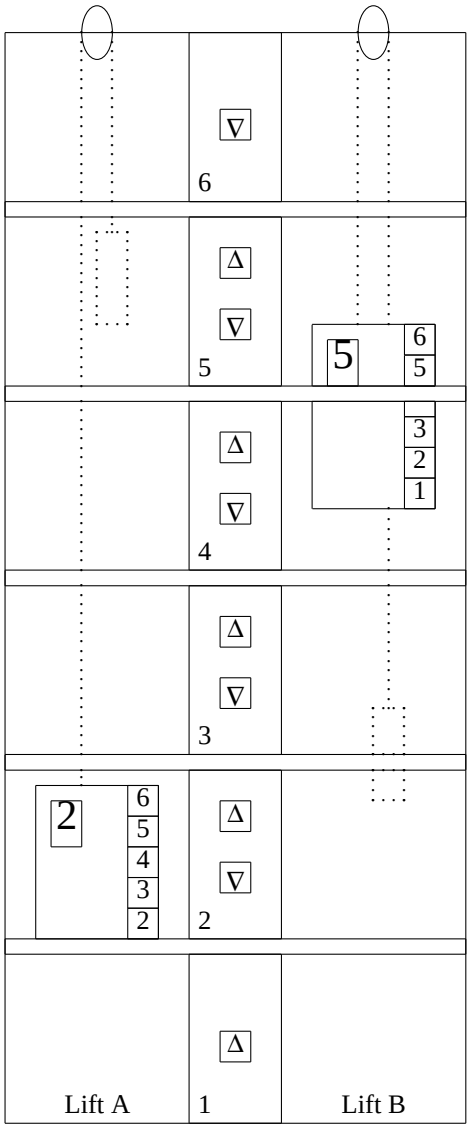

Figure 1: Liftshaft layout.

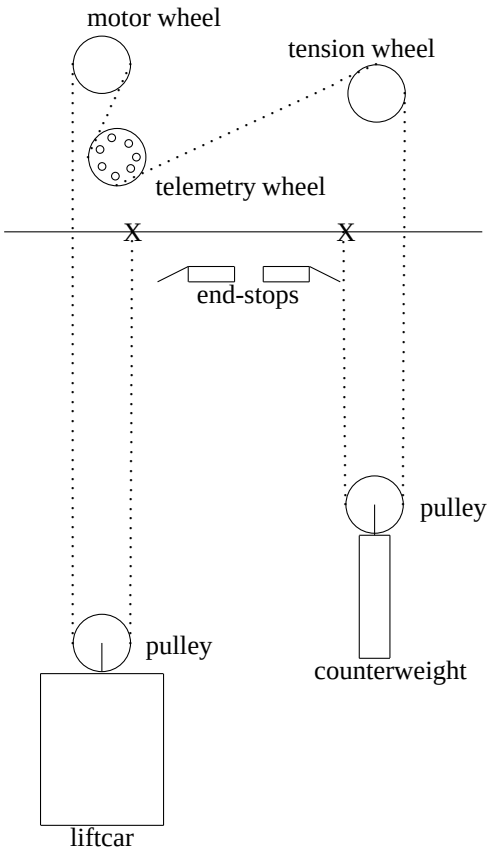

Figure 2: Lift-car mechanism.

4 Telemetry: an interrupt handler

To determine the position of each of the two liftshaft cars, a telemetry mechanism is used. The driving pulleys and strings incorporate a wheel in which a set of holes has been drilled. Two optical detectors straddle each of these wheels and generate values for status registers on the PIT devices indicating ''hole'' or ''not hole''. The PIT devices can be initialised to cause interrupts to the main processor every time this value changes.

A simple count of the number of telemetry interrupts, together with knowledge of the direction of motion, is enough to determine the exact position of either car to the accuracy of the distance the pulley string moves between holes (approximately 3mm), assuming that interrupts are dealt with in a timely fashion.

The telemetry process for each liftcar, A and B, is essentially identical (see Figure 3). Keeping a local record of the motor direction, current position, and next floor position, it awaits interrupts. When an interrupt occurs, it either increments or decrements the current position, depending on direction. At any time the process could

receive a message indicating a change of direction. As the lift car attains each new floor, it transmits a message to the appropriate lift controller to say so. Note that messages contain a value of type AB to determine the correct recipient of a telemetry or lift controller message.

5 Controlling and monitoring devices

It is often the case in embedded systems that although there are many sensors and effectors to be controlled, there are only a few interface devices available to do the controlling. The unit cost of devices, and size and weight considerations, are especially important for systems embedded in mass-produced goods. For this reason, there seem to be two common mismatches between the ideal level of hardware control and that which can actually be provided. Both these problems were encountered in programming the liftshaft.

1. *Overlapping.* Disparate mechanisms may not only share the same I/O device, but also the same registers in that device.
2. *Polling*. Although it may be desirable for certain sensors to generate interrupts, the actual devices available may not be able to provide as many interrupts as there are sensors, and polling has to be used.

5.1 Shared registers

The liftshaft motors, buttons, and call-lamps are controlled through two PITs, one for sensing, the other for effecting. Each device has two data registers which are connected to the apparatus. The different effectors and sensors overlap in these registers. Figure 4 shows the register bit-mapping on the PIT devices.

PIT device 2 is used for control: up and down refer to the call-lamps behind the buttons on the shaft. A and B are the motors for the two lift cars. PIT device 1 is used for the sensors. Here, the registers hold the instantaneous state of the call buttons and the end-stop sensors for the cars A and B. Similar sharing of registers occurs on the DUART device connected to the in-car buttons and LED displays.

This overlapping presents a problem, because `getReg` and `putReg` can read and write only whole-register values at a time. Single-bit operations must be implemented at some level as a word-read followed by a word-mask. But depending on the device, particular registers may be write-only.

So, when turning one motor on with `putReg`, a process must ‘‘know’’ the state of the other motor, and also of a couple of call-lamps, otherwise it could inadvertently cause them to change too. However, the program will control the motors and lamps from *separate* processes.

The solution is for a single server process to guard access to shared registers. Write-only registers are *shadowed* in the local state of the server process. Figure 5 shows such a server.

Other processes send an *updating function* in a message in order to communicate with the registers. Of course, functional abstraction hides the details from the calling process: each updating function is a bit masking operation, as shown in Figure 6. The functions are defined here in tabular style, because the mapping from logical action to control action is not orthogonal. If the system had a larger number of lifts, the hardware location of lifts in registers could probably be computed arithmetically from the logical action rather than by pattern-matching.

5.2 Polling vs. interrupts

It is important to realise how the call buttons and lamps are intended to work. When a button is pressed, the corresponding lamp should be lit immediately. The lamp should be extinguished only when the call is serviced by a lift. The set of lamps therefore acts as a persistent record of outstanding lift requests. Ideally, the interface logic should interrupt when a button is pressed, since such an event is infrequent by comparison with other events. Due to the shortage of interrupt lines on the PIT however, the hardware is not configured to make this possible. The only way to determine when a button has been pressed is by polling the data registers.

For these reasons, part of the expected behaviour of the buttons and lamps is programmed within the shared register server process. This process periodically reads the state of the buttons and copies it to the lamps *without* extinguishing any lamps already lit. This can be accomplished by a simple bit-wise OR operation. Provided the period is shorter than the average time for which a user holds a button, the visible behaviour should be as expected. Any other process needing to know whether a button has been pressed can simply request to see the current state of the lamps. Lamps are lit *only* by the shared register process; they are extinguished *only* by requests from other processes. This scheme is reflected in Figure 7, which extends the process definition of Figure 5.

To determine whether a lamp is lit or not, another process must first read the register state then apply a bit mask to it, as in Figure 8. A similar operation determines whether an in-car button request is still outstanding. Again, functional abstraction hides the details from the calling process.

```
data AB       = A | B
data TelemMsg = Change AB Dir
data Dir      = Down | Up | Stop
instance RecipientFor TelemMsg where
    address (Change ab _) = Telem ab

telemetry :: AB -> IntAction TelemMsg ()
telemetry ab =
    loopwith (updFlr 0 Stop) (\(pos,dir,flr) ->
        select (\Interrupt ->
                    let newpos = updpos dir pos in
                    if newpos == flr then
                         send (Reached ab (floor newpos)) >>
                         result (updFlr newpos dir)
                    else result (newpos, dir, flr))
               (\(Change _ newdir) -> result (updFlr pos newdir)))
    where
        updpos :: Dir -> Int -> Int
        updpos Stop n = n
        updpos Down n = n-1
        updpos Up   n = n+1

        updFlr :: Int -> Dir -> (Int,Dir,Int)
        updFlr pos dir = (pos, dir, nextfloor pos dir)
```

Figure 3: Telemetry with interrupts.

register/bit	7	6	5	4	3	2	1	0
pit2_padr	down4	down3	down2	up5	up4	up3	up2	up1
pit2_pbdr	X2	X1	Bon/off	Bdir	Aon/off	Adir	down6	down5
pit1_padr	down4	down3	down2	up5	up4	up3	up2	up1
pit1_pbdr	Q1	P1	Bbot	Btop	Abot	Atop	down6	down5

Figure 4: PIT effectors and sensors.

```
data ShRegT = UpdReg AB (Word->Word)
            | GetReg AB AB
instance RecipientFor ShRegT Pid where address = const SharedReg

sharedreg :: DevAction ShRegT ()
sharedreg =
    let initval = byte 0 in
    putReg (pit2 padr) initval >>
    putReg (pit2 pbdr) initval >>
    loopwith (initval,initval) (\(a,b) ->
        recv ? \msg ->
        case msg of
            (UpdReg A f) -> let fa = f a in
                            putReg (pit2 padr) fa >> result (fa,b)
            (UpdReg B f) -> let fb = f b in
                            putReg (pit2 pbdr) fb >> result (a,fb)
            (GetReg p A)-> send (Reg p a) >> result (a,b)
            (GetReg p B)-> send (Reg p b) >> result (a,b))
```

Figure 5: Shared register server.

*

```
motor :: Process action msg => AB -> Dir -> action msg ()
motor A Up   = send (Change Up) >>
               send (UpdReg B (setbit 3 . clrbit 2))
motor B Up   = send (Change Up) >>
               send (UpdReg B (setbit 5 . clrbit 4))
motor A Down = send (Change Down) >>
               send (UpdReg B (setbit 3 . setbit 2))
motor B Down = send (Change Down) >>
               send (UpdReg B (setbit 5 . setbit 4))
motor A Stop = send (UpdReg B (clrbit 3))
motor B Stop = send (UpdReg B (clrbit 5))

lampOff :: Process action msg => Floor -> Dir -> action msg ()
lampOff (Floor 1) Up = send (UpdReg A (clrbit 0))
-- etc.

writeLED  :: Process action msg => AB -> Floor -> action msg ()
clearCall :: Process action msg => AB -> Floor -> action msg ()
readInCarButtons :: Process action LiftMsg => AB -> action LiftMsg Byte
readShaftButtons :: Process action LiftMsg => AB -> action LiftMsg Word
-- etc.
```

Figure 6: Shared register update functions.

```
data ShRegT = as before, but adding
            | ClockTick

sharedreg =
    as before, up to
        case msg of
            as before, but adding
            ClockTick -> getReg (pit1 padr) ? \aval ->
                         getReg (pit1 pbdr) ? \bval ->
                         let fa = (notw aval) `orw` a
                             fb = (notw bval) `orw` b in
                         if fa==a && fb==b then
                              result (a,b)
                         else putReg (pit2 padr) fa >>
                              putReg (pit2 pbdr) fb >>
                              result (fa,fb)
```

Figure 7: Extension to shared register server.

```
lamplit :: AB -> Floor -> Dir -> Action LiftMsg Bool
lamplit me (Floor n) dir =
    let ab = whichreg dir n
        bm = bitmask  dir n in
    send (GetReg me ab) >> recv ? \(Reg _ val) -> result (mask bm val)

carcall :: AB -> Floor -> Action LiftMsg Bool
-- etc.
```

Figure 8: Reading a shared register value.

*

6 The lift scheduling algorithm

Now that various device drivers have been seen, the complete lift scheduling algorithm can be presented (see Figure 10). We treat each lift car, A and B, as an independent process with a local state. The state indicates whether the car is in motion; if so, whether it should stop at the next floor it reaches, and the current (or most recent) direction of motion and floor. A case analysis of the local state determines what action must be taken next: when the car is stationary, the decision is whether or not to start it moving, and if so, in what direction; when the car is in motion, but shortly before it reaches each new floor, the decision is whether or not to stop at the approaching floor.

The monadic I/O functions seen in Figures 6 and 8 are used to communicate with device drivers: to read requests from shaft and in-car buttons, and to write requests to motors, the LED display, and the lamps. The monadic function `checkRequests` determines whether there are any outstanding requests in a particular direction starting from a particular floor, first checking in the direction the lift has most recently been moving, then in the opposite direction. In either case if a request is found the lift is set in motion. When in motion, the car LED display is updated on reaching each new floor. When the car reaches the appropriate floor, it is stopped and the controller returns to check for new requests. If no request is found, the controller simply idles for a brief period of time before checking again.

7 Results

The liftshaft program and device drivers illustrated here fit comfortably on the described embedded architecture. Embedded Gofer's run-time system is about 48kb, and the compiled program is a further 82kb. The program uses a heap of 10,000 cells (about 80kb of data space), and the stack and look-up tables are about another 40kb. The total run-time memory usage is therefore around 250kb. This is small for a functional program, although an embedded systems programmer might still think it rather large.

Language	C	Modula	Gofer
Lines of code	2235	1082	770
Average function length	15.3	16.6	10.5

Figure 9: Code size comparisons.

Quantitative comparisons to earlier controller programs written in Modula and C can be made. The raw functional source code is smaller, although not by much (see Figure 9). The major reason for this is that the multitude of device register definitions and initialisations take the same number of lines, no matter what the language. Taking the average function length as a rough readability index, again the functional code has a definite, though slight, advantage. More subjectively, in comparison with the previous imperative solutions, we do claim improved readability: the lift controller of Figure 10 is clearer and more concise. We find it easier to follow because each function's result is dependent only on its arguments, and each process's actions are dependent only on its arguments and incoming messages/interrupts.

A further advantage of coding the embedded program in a functional language is the increased ability to parameterise objects. For instance, we have illustrated one block of code for two lift controllers – the code is parameterised on the lift name, A or B. In both the Modula and C solutions, the controller code is duplicated (with minor changes) for each lift.

Telemetry interrupts occur at up to 20Hz; at present the program and run-time system are fast enough to run a single lift. However, when both lifts are in motion, one of them is inclined to overshoot its destination floor. This suggests that the implementation has exceeded its maximum throughput of interrupts; compared with a single lift system, twice as many interrupts arrive per unit time, but the same amount of evaluation is required for each interrupt.

Part of the reason for this failure is severe inefficiency in the implementation of the incremental garbage collector: up to 40% of runtime is expended on memory management! The second factor limiting the speed of the application is that Gofer is essentially an interpreted language system: the evaluator takes about 15% of runtime. Several optimisations are already planned here which should soon allow the second liftcar to run simultaneously without the loss of accurate positioning.

8 Related work

One area of computing practice that has so far been largely incompatible with the introduction of functional languages is embedded systems: the computer hidden inside another machine. Most functional languages are designed to run on large, general-purpose computers that typically have a screen, keyboard, and file store.

Input and output (I/O) to and from these devices is usually provided either in the language itself, or through privileged library routines. However for some applications, notably control systems, this characterisation of the need for I/O is totally inadequate. In every different system, the computer must control different transducers, and read different sensors. In other words, the specific I/O requirements are unique to each application, and cannot be part of a standard library.

Embedded Gofer differs from previous functional languages for embedded systems in several respects.

```
data HowFar   = ToNext | Beyond
data Motion   = Moving HowFar | Stopped
data LiftMsg  = Reached AB Floor | Reg AB Word
type CarState = (Motion,Floor,Dir)

instance RecipientFor LiftMsg where
    address (Reached ab _) = Lift ab
    address (Reg ab _)     = Lift ab

lift :: AB -> Action LiftMsg ()
lift ab = writeLED ab (Floor 1) >>
          loopwith (Stopped, Floor 1, Up) lift'
  where
      lift' :: CarState -> Action LiftMsg CarState
      lift' (Moving ToNext, _, dir) =
              recv ? \(Reached _ flr) ->
              motor ab Stop >> writeLED ab flr >>
              wait briefpause >> result (Stopped, flr, dir)
      lift' (Moving Beyond, _, dir) =
              recv ? \(Reached _ flr) ->
              writeLED ab flr >>
              let nf = next flr dir in
              stopAt nf dir ? \howfar -> result (Moving howfar, nf, dir)
      lift' (Stopped, flr, dir) =
              checkRequests flr dir ? \ndir ->
              case ndir of
                   Stop -> wait briefpause >> lampOff flr dir >>
                           result (Stopped, flr, dir)
                   _    -> clearCall ab flr >> lampOff flr ndir >>
                           send (Change ab ndir) >> motor ab ndir >>
                           let nflr = next flr dir in
                           stopAt nflr ndir ? \howfar ->
                           result (Moving howfar, nflr, ndir)
      stopAt :: Floor -> Dir -> Action LiftMsg HowFar
      stopAt flr dir = carcall ab flr  ? \x ->
                       lamplit ab flr dir ? \y ->
                       result (bool2howfar (x||y))
      next :: Floor -> Dir -> Floor
      next (Floor n) Up   = Floor (max (n+1) top)
      next (Floor n) Down = Floor (min (n-1) gnd)

      checkRequests :: Floor -> Dir -> Action LiftMsg Dir
      checkRequests flr dir =
              readInCarButtons ab ? \incar ->
              case whichDir (inCarMask flr) dir incar of
                  Up   -> result Up
                  Down -> result Down
                  Stop -> readShaftButtons ab ? \shaft ->
                          result (whichDir (shaftMask flr) dir shaft)
      whichDir :: (Dir -> Word) -> Dir -> Word -> Dir
      whichDir mask dir word =
          let otherdir = rev dir in
          if (mask dir) `orw` word then result dir
          else if (mask otherdir) `orw` word then result otherdir
          else result Stop
```

Figure 10: A lift scheduling algorithm.

*

Unlike Erlang [AVW93], it has a strong polymorphic type system, it can handle interrupts, and most importantly, it allows device drivers to be written within the functional framework. The Erlang book [AVW93] presents a solution to the liftshaft scheduling problem but shows no device drivers, because they must be written in a different language.

Embedded Gofer's process model improves upon previous models by being both type secure (unlike e.g. [Stoye86]) and referentially transparent (unlike e.g. [JH93]). Previous functional treatments of interrupts have either followed a narrow termination semantics (e.g. [Gordon94]), or else have been largely undeveloped (e.g. [Clack89]), whereas Embedded Gofer's treatment is both more realistic and has been tested in practice. An earlier design of Embedded Gofer [WR95] treated I/O in continuation style rather than monadically as here; typechecking of messages [WR94] has since been added.

Finally, Embedded Gofer has been designed explicitly for use on machines with small memory, a stated aim which is absent from every other language design referred to here. A full account of Embedded Gofer appears in the first author's doctoral thesis [Wallace95].

9 Future work

Some inefficiencies in both the language system implementation and the application have already been identified. It is hoped that when time profiling techniques for lazy functional languages [SPJ95] are sufficiently developed, their use will highlight further areas for improvement. Also, although space-profiling is now a well established technology [RW93], its absence from Embedded Gofer has caused certain questions about the memory requirements of applications to be left unanswered. Research into the specific patterns of memory usage found in embedded applications could eventually allow the embedded systems designer a much greater degree of control over the price/performance tradeoff.

The current scheme for typechecked message-passing has advantages over previous mechanisms, but still requires a form of address to be attached to some messages. It is possible that new techniques could be found to reduce this burden on the programmer still further.

Real-time expression is achieved at the moment only by programming a timer device. By integrating certain timing-related monadic operations into the language design, the full weight of schedulability analysis [Burns91] could be brought to bear on embedded functional programs, potentially simplifying their runtime characteristics at the same time as providing behavioural guarantees to the programmer.

It would be pleasing to be able to apply formal reasoning techniques to entire embedded programs. Although individual Embedded Gofer processes are amenable to equational reasoning and transformation, a formalism for describing interaction between processes is an important goal for future research.

10 Conclusions

Although advanced software techniques such as lazy higher-order functional programming have existed for many years, very few applications yet make adequate use of them. One reason is that there are still some technical problems in integrating such techniques with existing technologies. Embedded Gofer, making use of recent advances such as monadic I/O and new type systems, considerably widens the application area of functional languages.

In particular, Embedded Gofer allows the description not only of *what* values are to be used for I/O, but *how* they are to be transferred. Device drivers can be written *in the functional language*, where previously they had to be written externally. One high-level language can be used *throughout* a system. Functional abstraction allows the application program to remain concerned only with the value level, whilst device drivers deal with the *how*.

Several examples of architectural compromises were seen in the liftshaft problem. Disparate controllers and sensors shared the same I/O registers. Some devices used polling where interrupts would have been preferable. The device drivers were able to cope with these deficiencies, and hide them from the application program.

It is likely that embedded systems will form an ever more important part of the computing industry's range of products in the coming years. The more quickly that software can be written for such products, the better competitive advantage a company will derive, provided that software is correct. The more confidence that can be placed in the correctness of embedded software, the better will end-users of those products be protected from frustration, danger, and perhaps fatality.

Functional programming has considerable potential in this field. It offers rapidity of development [HJ95], and reduction of undetected errors through the use of higher-order combining functions, strong polymorphic type systems, automatic memory management, and the property of referential transparency.

Acknowledgements

This work was supported by a research studentship to the first author from the Department of Education for Northern Ireland. Additional support was provided by Canon Research Centre Europe Ltd. The authors would like to thank Mark Jones for his commitment to users and modifiers of his Gofer programming system, Rick Pack for supplying us with hardware advice and fixes, and several anonymous referees for useful comments leading to improvements in this paper.

References

[AVW93] **Concurrent programming in Erlang**, Joe Armstrong, Robert Virding, Mike Williams, Prentice-Hall (1993)

[Burns91] **Scheduling hard real-time systems: a review**, Alan Burns, Software Engineering Journal **6***(3)*, pp.116-128 (May 1991)

[Clack89] **Signal handling for functional programs**, Chris Clack, University College London (May 1989)

[Gordon94] **A proposal for monadic I/O in Haskell 1.3**, Andrew D Gordon, Cambridge University Computer Laboratory (March 1994)

[HJ95] **Haskell vs. Ada vs. C++ vs. Awk vs. ..., An experiment in software prototyping productivity**, Paul Hudak, Mark P Jones, Journal of Functional Programming **5***(to appear)* (1995)

[JH93] **Implicit and explicit parallel programming in Haskell**, Mark P Jones, Paul Hudak, Department of Computer Science, Yale University, **YALEU/DCS/RR-982** (August 1993)

[Jones93] **A system of constructor classes: overloading and implicit higher-order polymorphism**, Mark P Jones, pp.52-61, in *Proceedings of 6th Conference on Functional Programming Languages and Computer Architecture*, Copenhagen, ACM Press (June 1993)

[Jones94] **The implementation of the Gofer functional programming system**, Mark P Jones, Department of Computer Science, Yale University, Technical Report, **YALEU/DCS/RR-1030** (May 1994)

[PJW93] **Imperative functional programming**, Simon Peyton-Jones, Phil Wadler, in *Proceedings of 20th Symposium on Principles of Programming Languages*, Charleston, South Carolina, ACM Press (January 1993)

[RW93] **Heap profiling of lazy functional programs**, Colin Runciman, David Wakeling, Journal of Functional Programming **3***(2)*, pp.217-245 (April 1993)

[SPJ95] **Time and space profiling for non-strict higher-order functional languages**, Patrick Sansom, Simon Peyton Jones, pp.355-366, in *Proceedings of Symposium on Principles of Programming Languages*, San Francisco (January 1995)

[Stoye86] **Message-based functional operating systems**, William Stoye, Science of Computer Programming **6***(3)*, pp.291-311 (May 1986)

[WR93] **An incremental garbage collector for embedded real-time systems**, Malcolm Wallace, Colin Runciman, in *Proceedings of the Winter Meeting*, Chalmers University of Technology, Gothenburg, Sweden, **PMG-R73** (June 1993)

[WR94] **Type-checked message passing between functional processes**, Malcolm Wallace, Colin Runciman, in *Proceedings of the Glasgow Functional Programming Workshop*, Springer Verlag, BCS Workshops in Computer Science (Sept 1994)

[Wallace95] **Functional Programming and Embedded Systems**, Malcolm Wallace, in *DPhil Thesis*, Dept. of Computer Science, University of York, UK (January 1995)

[WR95] **Extending a functional programming system for embedded applications**, Malcolm Wallace, Colin Runciman, Software Practice and Experience **25***(1)* (January 1995)