EMBEDDED FUNCTIONAL PROGRAMMING IN HUME

Gergely Patai and Péter Hanák
Department of Process Control and Information Technology
Budapest University of Technology and Economics
Budapest, Magyar tudósok körútja 2., H-1117, Hungary
email: {patai,hanak}@iit.bme.hu

ABSTRACT
Embedded systems represent a rapidly growing branch of information technology, characterised by the need for increased dependability, timeliness and efficiency. While functional languages allow developers to produce highly reliable and maintainable code, they ignore the aspect of time, and their efficiency is inferior to those of low-level languages currently dominating this field. Hume is a novel hybrid language that combines the functional paradigm with ideas from hardware design. It focuses on producing time- and space-constrained code while keeping the benefits of programming on a high level. This paper describes the development of a lift control simulation in Hume running on low-performance embedded hardware platforms, including the process of porting the code to these platforms.

KEY WORDS
Programming Tools and Languages, Embedded Systems, Functional Programming, Hume

1 Introduction

Embedded systems (ES) represent an important area of computing; their significance is expected to grow rapidly in the years to come. As opposed to other programmed systems, their operation is autonomous, receiving input from sensors and sending output to actuators or transducers—in the broad sense of these words—, and in a typical setting they are programmed to perform a specific task.

An ES has to provide a reliable service for extended periods of time, often without human intervention, while living on minimal resources. The majority of these systems is programmed in low-level languages because of the harsh performance constraints, which makes it very hard to ensure their correctness and safety.

The demand for ES applications is increasing, while their time to market is decreasing. Consequently, developers have less and less time to spend on a project. On the other hand, it is impossible to train enough new experts, because ES development requires above average design and programming skills. Given the typically strict dependability requirements of these systems, compromising quality is not a viable option. Developers need better tools.

Functional programming is a paradigm worth considering in this context, since it comes with the promise of accelerating development while enforcing a high quality of code, all at the price of losing efficiency. However, as long as a functional program can comply with the specifications for a given task, this loss is irrelevant, and the developer only benefits from using a functional language.

We present an example embedded application, a lift controller, implemented in the hybrid functional language Hume [1][2], and compare its efficiency to an equivalent C implementation. The example was first presented in [3].

2 Requirements for ES Programming

Developing ES usually means extra difficulties in comparison to developing for the desktop, affecting the final products, as well as the development environments. [4] [5]

The product has to meet most of these requirements:

- **reliability**: the system must be guaranteed to work without intervention for a long time
- **robustness**: input can have extreme variations; error handling must cover literally every physically conceivable constellation of events and in certain cases ensure continuous operation in all circumstances.
- **limited response time**: the system must react to signals within a short period of time specified already during the design phase. In general, an ES is expected to be deterministic.
- **scarce resources**: severely limited amount of memory (possibly as little as a few kilobytes), slow processor (its clock rate might be as low as a few megacycles), and very little power available for use.
- **concurrency**: many tasks naturally break up into concurrent processes that often have to communicate with each other without interfering with others' internals.

On the development side there are other new problems:

- **need for cross compilation**: the development platform is separated from the target platform; migrating the code to the target is by no means a trivial process.
- **debugging with emulators**: debugging is very hard, if not impossible, without a proper emulator; however, emulation tools still have limited capabilities.

- **early design decisions**: resource constraints must be considered early in the design process.
- **operating systems**: embedded applications often lack an OS at the bottom. While OS are available even for heavily constrained systems, they are not comparable to desktop OS in terms of features and security.

At the same time, we also have to consider the usual tenets of software development: maintainability, modularity, code reuse when applicable, design on the whole, testing, ease of programming and so on. While these aspects are generally important in every non-trivial system, the new difficulties outlined above make it even more desirable to be able to handle them as smoothly as possible.

3 Pros and Cons for Functional Languages

Here is a list of important aspects to be considered when we use a functional language to program an ES:

- **fitting applications**: while functional languages are not always the most appropriate choice, many problems have a concise functional representation.
- **design**: a high-level language is a kind of executable design. Functional languages definitely have a place in the early phases of development even if the production code will be created in a low-level language.
- **reliability**: by providing a compact and easily tractable representation of an algorithm, the programmer has less chance to get it wrong, and interesting properties (e.g. invariants) can be proved more easily.
- **concurrency**: functional programs are composed of greatly isolated, independent and thread-safe, smaller units, suitable for concurrent execution.
- **guaranteed time**: the functional paradigm does not address timing, only correctness. Implications of multitasking with process priorities are irrelevant to the language used.
- **performance**: functional programs are usually slower than their imperative equivalents, but the difference is often negligible because of the powerful processors.
- **learning time**: most programmers are trained in imperative programming, and a proper adjustment can take a long time. On the other hand, functional programming education at universities has a long tradition.

Summing up, using traditional functional languages in ES development means sacrificing raw performance, and also losing of precise control over what and when should happen. What we get in exchange is improved correctness, maintainability and faster development.

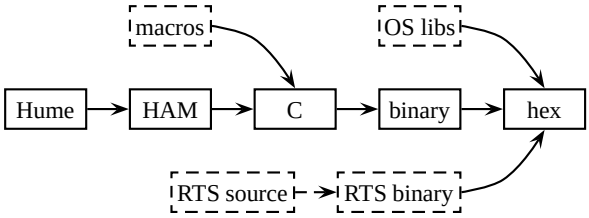

Figure 1. Producing an executable from Hume source

We have chosen the Hume language to demonstrate the advantages of functional programming for ES.

Hume, a hybrid functional language, was specifically designed to address the issues of performance, time and space constraints, and controllability, while keeping the advantages of functional programming. The key idea is to separate aspects related to *expression* and *coordination*.

A Hume program is analogous to a hardware description of a sequential network which consists of solely combinational *boxes* (in other words, stateless transformations which are evaluated, at least in theory, faster than the global clock rate) with all of their inputs and outputs connected to registers. The boxes are represented by pure functions, while the registers between them are called *wires* in Hume; in other words, the wires are where the state of the system is stored. The wiring determines dynamic behaviour (coordination), while the boxes are responsible for side-effect-free data manipulation and transformation (expression).

The concept of boxes and wires makes the implementation of finite state machines, like lift control logic, easy in Hume.

4 Porting Hume to Target Platforms

One of the options to run Hume programs is compiling them to native code. This is performed by translating them to C and let the C compiler produce the binary. The Hume runtime system (RTS) does not use any special resources, i.e. dynamic memory allocation or threads, and it is not sensitive to endianness, therefore it is very easy to port. The only real difficulty lies in accessing the peripherals of the target system from Hume. Fortunately, it is possible to fully customise the code generated for I/O boxes by preparing a set of macros for each platform; there is no need to modify the compiler itself. The final compilation process is shown on Figure 1.

The diagram uses dashed lines to denote the components and steps which belong to the porting process. These are one-time operations; in order to get a Hume program to compile, we needed to prepare macros and adapt the RTS, whose source is available in C. The RTS is compiled only once, producing object code that will be linked to the user generated code at each compilation along with the OS runtime. It is possible to perform the final linking step on-site, e.g. using the code deployment facilities of the Contiki embedded OS [6].

In the lift example we chose the simplest solution to bind peripherals to Hume output boxes: the custom macros generate code that allow us to directly call C functions as special outputs. Output functions must be declared as `void fn_name(HEAP *ptr)`, where `HEAP` is a data type declared by the RTS, and their implementations must be statically linked to the final executable code. The names of the functions are specified in the Hume source. For instance, the declaration

```
stream numberDisplay to "number_display";
```

requires the function

```
void number_display(HEAP *ptr) {...}
```

to be present somewhere in the runtime.

If everything is prepared, Hume source can be compiled and run on the target, which is shown by the middle line of the figure. The steps are the following:

1. The Hume source is compiled to HAM (Hume Abstract Machine) code with `phamc` (Prototype HAM compiler). [7]

2. The HAM code is transformed to C by `ham2c`, which needs the macros to be present.

3. The resulting C code can be compiled into a binary.

4. The linker combines the newly produced binary with the runtime and emits an executable, in our case in Intel hex format, which can be uploaded to the mote.

Despite the slowness of a typical embedded CPU, using an interpreter can be still a viable option, because bytecode is easier to manage (e.g. hotload, monitor) than native code, and also safer to execute. Besides, bytecode is normally more compact than native code, which can become an important factor in certain scenarios, e.g. in a sensor network where minimising communication load is desirable. On the other hand, interpreting is also more CPU intensive, which means increased power consumption on the long run. This trade-off must be considered for each new task.

The interpreter is built on top of the RTS, hence it does not require much additional code memory. While there is no bytecode defined for HAM, it is possible to simply pre-parse a source file and populate the data structures used by the interpreter to prepare a distributable package.

5 Lift Control Design

The application is a single lift that can move between floors as the passengers request: calling from outside, specifying the direction they want to go and sending to a specific destination floor from inside. This system consists of three entities: cabin, door and requests.

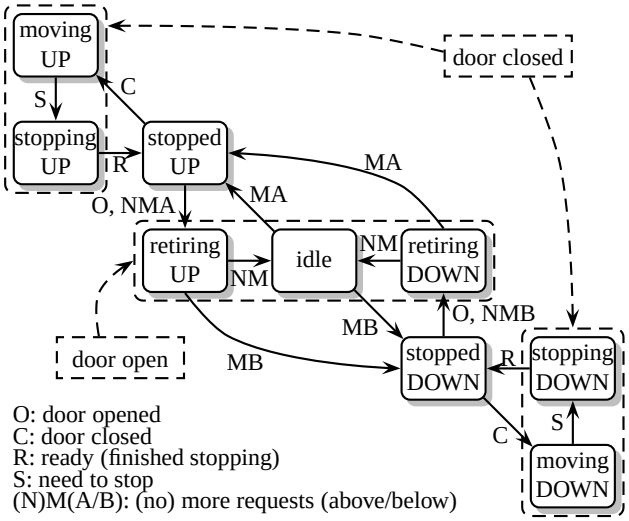

Figure 2. Lift cabin movement states

5.1 Cabin

Cabin movement is described by the well-known elevator algorithm, starting from an idle state:

1. When a request is made on some floor, start moving towards it.

2. Stop on floors with a request in the same direction.

3. Keep the direction until the cabin reaches the last floor with any request.

4. If there are requests in the other direction, start servicing them and repeat from step 2; otherwise enter idle state and go to step 1.

In order to achieve this behaviour, we need to introduce nine internal states, as shown on Figure 2.

5.2 Door

A door is usually considered to have two states: open and closed. Since physical doors, especially lift doors need considerable time to move between these states, other relevant events might occur during the transitional phases. Therefore, we added a state for each direction, ‘opening’ and ‘closing’, as seen on Figure 3.

5.3 Requests

Passengers control the lift by sending requests either from inside or outside. When inside, they can signal their intention to get off at a specific floor, while on the outside they tell the system which direction they want to go.

There are three kinds of requests to be recorded for each floor: ‘out’, ‘up’ and ‘down’. However, the first can be substituted with the combination of the other two, because

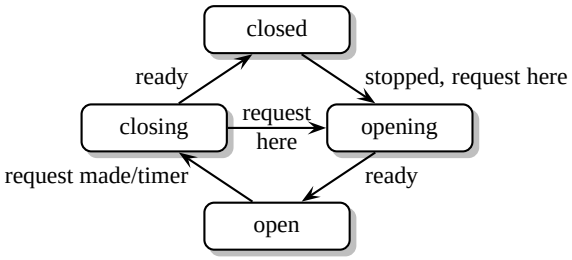

Figure 3. Lift door states

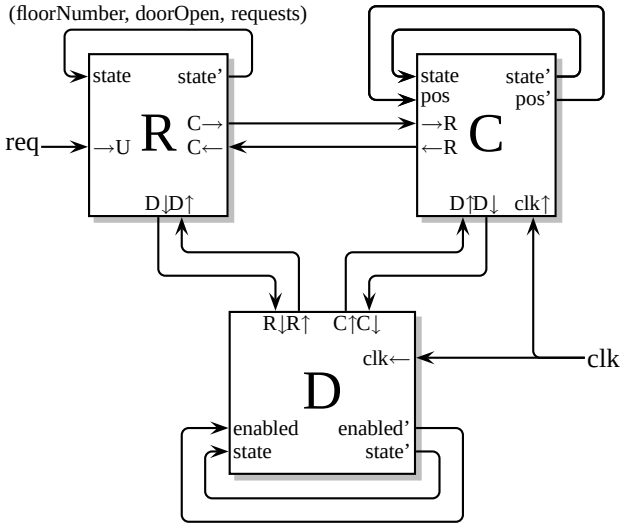

Figure 4. Structure of the Hume implementation

a simultaneous upward and downward request is enough to stop the lift. The requests are stored in a component called 'request manager', which contains a memory element that holds two bits of information for each floor.

5.3.1 Coordination

The interaction of the main components is required for many state transitions. This can be organised either using a central control unit or by letting everyone communicate directly to others. Since in this system we have only three components, there is no need for additional hierarchy.

6 Lift Control Simulation in Hume

Hume offers a direct translation of the above concept into code due to the visual nature of the boxes and wires and the declarative rules of transition inside the boxes.

6.1 Interconnections

The coordination language of Hume is essentially visual, depicting the global structure of the program as a set of boxes connected with wires. The image of interconnections can be mechanically transformed into Hume syntax. The source is structured as follows:

```
program

... datatype declarations ...
... auxiliary functions ...

box requests
in (state::sRequests, fromUser::request,
    fromCabin::message, fromDoor::message)
out (state'::sRequests, toCabin::message,
     toDoor::message)
match
... 5 rules for the request manager ...

wire requests
  (requests.state' initially ...,
   timer.input, cabin.toRequests,
   door.toRequests)
  (requests.state, cabin.fromRequests,
   door.fromRequests);

... cabin box and wiring ...
... door box and wiring ...
... auxiliary box (user input + timer) ...
```

The final control logic code corresponds to Figure 4. The door and cabin boxes have two state variables (those that are fed back in a loop), because the notion of state as introduced above is kept separate under the name 'state'. In the case of the cabin the current position is an independent state variable, while the 'enabled' property of the door is in reality a function of the cabin's state (it is false if the cabin is idle or currently deciding whether to reverse the direction of movement, in order to prevent the door from closing). The state of the request manager is a triple composed of the requests, a boolean indicating whether the door is open and the current floor. Since it has no state transitions similar to those of the other two components, we decided not to separate these pieces of data.

6.2 The Request Manager in Detail

To show how Hume can be used in such an application, let us have a look at the request manager component.

The internal state of this component maintains a copy of the cabin's position and the state of the door, both sampled at the appropriate moments. This helps reducing the communication with the other components, thereby making the protocol and consequently the whole system considerably simpler.

The behaviour of the request manager can be described with five simple rules. Since this box has four input and three output wires (including the state loop), each rule is a mapping of a quartet to a triple:

```
(state, fromUser, fromCabin, fromDoor) ->
                (state', toCabin, toDoor)
```

Here, the wire `state` contains the triple `(floorNumber, doorOpen, requests)`, the wire `fromUser` carries requests encoded as simple integers, and the wires between the three components can only hold data of the type called `message`, a tagged union, which encodes all the possible messages in the system. The rules of the request manager are the following:

1. If the cabin asks whether there are any requests ahead, send a boolean answer; the internal state is not affected at all. The question contains the direction of the cabin's movement, which is needed to determine the answer. The corresponding code snippet is as follows; these lines immediately follow the `match` keyword in the declaration of the `requests` box:

   ```
   ((f, o, rs), *, MsgAskAhead ad, *) ->
     ((f, o, rs), MsgRequestAhead
                  (isAhead rs f ad), *) |
   ```

   The rule describes a mapping of four inputs to three outputs. The first input is the state information, which is immediately broken down into its components (`f` is the floor number cache, `o` contains the fact whether the door is open and `rs` is the vector that contains the requests). The second input is coming from the user. The asterisk is used to denote that the rule ignores the wire and leaves its contents intact. The third input is the message wire coming from the cabin; this rule can only fire if the incoming message has the `MsgAskAhead` tag, which is determined by simple pattern matching. The fourth input is the message from the door, which is also irrelevant for this rule.

   The output wires can be described similarly. The state does not change, so it is fed back without modifications. The second output is the wire going to the cabin, therefore the message is pushed onto it. The third output belongs to the door: the asterisk has a similar meaning as on the input side, i.e. the wire is ignored—no output is needed—as far as the current rule is concerned.

2. If the cabin asks whether the current floor needs to be serviced, cache the floor number and answer. The caching is done by replacing the value of the floor number in the outgoing state wire by the one supplied in the question. This is the right moment to make the copy, since it is the first moment at any given floor, and floor numbers are not relevant while the cabin is moving between two stops. The previously cached floor number is discarded.

   ```
   ((_, o, rs), *,
    MsgAskCurrent af ad, *) ->
     ((af, o, rs), MsgNeedToStop
            (needToStop rs af ad), *) |
   ```

3. If the door says it has opened, ignore requests for the current floor until further notice. The `doorOpen` member of the state is set to `true` regardless of its previous value. Also, this is the point where requests are cleared for the current floor.

   ```
   ((f, _, rs), *, *, MsgOpened) ->
     ((f, true, delRequest rs f), *, *) |
   ```

4. If the door says it started closing, enable all requests again. This is done by setting `doorOpen` to `false`.

   ```
   ((f, _, rs), *, *, MsgClosing) ->
     ((f, false, rs), *, *) |
   ```

5. If a new request arrived and is not ignored (which happens only if it targets the current floor and the door is wide open), store it and notify either the cabin or the door depending on the relative position.

   ```
   ((f, o, rs), nr, *, *) ->
     let rp = relpos nr f in
       if o && rp == None then
         ((f, o, rs), *, *)
       else let nrs = addRequest rs nr in
         ((f, o, nrs),
         if rp != None then
           MsgNewRequest rp else *,
         if rp == None then
           MsgNewRequest None else *);
   ```

   Since this is the last rule, it is followed by a semicolon.

7 Comparison with C

7.1 C Implementation in the *mitmót*[1]

The C version is based on the Hume implementation; the transformation was a straightforward process. There are three kinds of components in the Hume implementation: boxes, 'message' wires (wires connecting boxes) and 'state' wires (feedback loops). They can be represented in C as shown in Table 1. Additional data manipulation is performed by simple functions in both languages.

functionality	Hume	C
logic	boxes (rules)	threads (algorithm)
state	state wires	local variables
communication	message wires	message boxes
auxiliary	functions	functions

Table 1. Components in Hume and C

7.2 Quantitative Properties

Both versions took about the same net time to code after finishing the design: a few hours. The Hume code was practically correct as soon as it was finished, we only needed to add some consume-and-ignore patterns (`_*`) to

[1]The first platform of our choice was a mote called *mitmót*, developed at the Department for Measurement and Information Systems, Budapest University of Technology and Economics, see http://bri.mit.bme.hu.

the left hand side of certain rules to resolve some blocking issues and add a message or two we forgot about. The C version took even less time to fix after completing the first running version; we only needed to adjust some messages due to copy-paste errors. Table 2 lists some quantifiable properties of the programs.

metric	**Hume**	**C**
lines of code*	314/265	682/650**
number of characters	9066	12000
comment frequency	everywhere	at declarations
executable size***	115212	35056
estimated own size	80412	6136

* with/without comment
**467 without curly brace lines
*** in bytes; OS adds 28920 bytes, RTS further 5880 bytes

Table 2. Quantitative comparison of Hume and C sources

While the C program was never run on anything but the *mitmót*, the Hume program was mostly tested on the PC and instrumented to display a meaningful message for each rule that fired.

Memory proved to be a more stringent limit than CPU power. This early version of the compiler cannot generate efficient code, because it does not use type information at all (the compiled code is practically untyped), and the HAM to C transformation is also mechanical. Only some structural optimisation is performed when the HAM code is generated. In the end, the C code is about 13 times smaller after deducting the OS and the RTS as well for the Hume executable.

Speed was no problem at all. On the other hand, we reached the limits of the available program memory.

8 Conclusions

The Hume approach provides the developers with a refined decision: instead of partitioning the task only into hardware and software components, the software half can also be further divided into high and low-level parts. There is a long way ahead to see where the functional paradigm can truly rival the current methodologies.

9 Future Work

Hume is still in the early phase of its evolution, therefore there are many directions to explore: graphical editor with code generation, optimising compiler, compilation vs. interpretation on the hardware platforms etc.

We are primarily interested in embedded applications with tight resource constraints. We are experimenting with other mote platforms (e.g. Tmote Sky), other operating systems (Contiki, TinyOS), and Hume code interpretation on the mote. This work is related to the RUNES project[2]. This project builds an example sensor network, which is supervised by a server programmed in Erlang [8], another functionally based programming language suitable for concurrent programming. The sensors—Tmote Sky units—are currently programmed in C, and therefore cannot be validated together with the program parts written in Erlang. Our aim is to program the sensors in Hume instead of C, and make all parts of the system appropriate for analysis, emulation and validation.

[2]Reconfigurable Ubiquitous Networked Embedded Systems, IST-2002-2.3.2.5, see http://www.ist-runes.org

10 Acknowledgements

The authors express their thanks to dr. Greg Michaelson and his team for the Hume programming tools and documentation, dr. Csaba Tóth and his team who provided us a *mitmót* platform to make experiments, and Zoltán Theisz (Ericsson Hungary) for turning our attention to RUNES, Tmote Sky and the Contiki OS.

References

[1] G. Michaelson and K. Hammond, The Hume Report, Version 1.1, 2006

[2] K. Hammond, and G. Michaelson, Hume: a Domain-Specific Language for Real-Time Embedded Systems, *In Proc. Conf. Generative Programming and Component Engineering (GPCE '03)*, *Lecture Notes in Computer Science*, (Berlin Heidelberg New York: Springer-Verlag, 2003)

[3] G. Patai, Programming Embedded Systems in Functional Languages, Budapest University of Technology and Economics, 2006

[4] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Approach*, (John Wiley and Sons Inc., 2002)

[5] A.S. Berger, *Embedded Systems Design: An Introduction to Processes, Tools, and Techniques*, (CMP books, 2002)

[6] A. Dunkels, B. Grönvall, T. Voigt, Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors, *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, 2004

[7] H-W. Loidl and K. Hammond, Specification and Cost Model for the Hume Abstract Machine, *Deliverable of the IST-510255 EmBounded project*, 2006

[8] J. Armstrong, R. Virding, C. Wikström, M. Williams, *Concurrent Programming in Erlang (second edition)*, (Prentice Hall, 1996)