Chapter 1

Is it Time for Real-Time Functional Programming?

Kevin Hammond[1]

Abstract This paper explores the suitability of functional languages for programming real-time systems. We study the requirements of real-time systems in general, outline typical language approaches for this domain, consider issues relating to memory and time usage, and explore how existing functional languages, including our own language Hume, match these requirements. We conclude by posing some research challenges that functional language designs and implementations must meet if they are to be regarded as suitable vehicles for real-time systems implementation.

1.1 INTRODUCTION

Functional programs use large amounts of memory. Functional programs are slow. It is impossible to predict memory and other resource usage for functional languages. Clearly, functional languages are therefore unsuitable for use in restricted memory settings with strong time requirements. Or are they? This paper explores the suitability of functional language designs for use in settings with strong limitations on resource usage such as real-time systems. It compares current functional approaches, including our own Hume notation (Section 1.5), with those used by other language paradigms, and outlines some challenges for functional language designs and implementations that must be met if functional programming is to be used for serious real-time programming.

[1]School of Computer Science, University of St Andrews, North Haugh, St Andrews, Scotland, KY16 9SS. **email:** kh@dcs.st-and.ac.uk.
This work has been supported by UK EPSRC grant GR/R 70545/01.

1.2 WHAT IS REAL-TIME PROGRAMMING?

The key characteristic of a real-time system is that its correctness depends not only on its functional behaviour, but also on the (real-)time or times at which it produces those results [9]. Such systems can be classified as having either soft real-time or hard real-time properties. Soft real-time has been defined as a situation where "nothing really serious happens if a time constraint is not met" [3]. Examples of soft real-time systems might include computer games, telephone switches, digital set-top boxes, or digital sound cards. In contrast, hard real-time involves guaranteed system response, and is often associated with safety-critical systems or ones with high penalty cost for failure. Examples include avionics control software, autonomous vehicles, or software used by stock market traders. In many situations, such as embedded systems, such real-time constraints are combined with other resource restrictions including memory limitations and even power consumption requirements. Despite the focus on real-time, such systems need not necessarily be ultra high-performance. The problem is to design systems that are sufficiently reliable, have minimal cost, and acceptable performance. Doing so in a cost-effective manner is a major bonus.

1.2.1 The Importance of Real-Time Systems

Real-time systems have been growing in importance in recent years. Numerically, a very high percentage of all computer systems produced today have real-time characteristics. Many of these are *embedded systems*. Real-time embedded systems are a fundamental part of modern everyday society in the shape of vehicle control systems, mobile telephones, GPS and consumer appliances such as DVD players, or digital set-top boxes. These commonplace devices are additional to those used in telecommunications, to promote automation in factories, to ensure security and safety in the home and workplace, to increase the safety and efficiency of transport and service industries, and for military uses etc. In fact, today more than 98% of all new processors are used in such systems [41].

1.2.2 Essential Properties of Real-Time Languages.

McDermid identifies a number of essential or desirable properties for a language that is aimed at hard real-time systems [31].

- *determinacy* – the language should allow the construction of determinate systems, by which we mean that under identical environmental constraints, all executions of the system should be *observationally equivalent*;
- *bounded time/space* – the language must allow the construction of systems whose resource costs are statically bounded – so ensuring that *hard real-time* and *real-space* constraints can be met;
- *asynchronicity* – the language must allow the construction of systems that are capable of responding to inputs as they are received without imposing total

ordering on environmental or internal interactions;

- *concurrency* – the language must allow the construction of systems as communicating units of independent computation;
- *correctness* – the language must allow a high degree of confidence that constructed systems meet their formal requirements [1].

These requirements may be relaxed to acceptable engineering tolerances for soft real-time systems. Moreover, the language design must incorporate at least:

- *periodic scheduling* to ensure that real-time constraints are met;
- *interrupts and polling* to deal with connections to external devices.

1.3 LANGUAGES FOR PROGRAMMING REAL-TIME SYSTEMS

Programming languages for real-time systems may be either specially designed to meet the requirements of the domain (domain-specific languages), or adapted from commonly used designs. Since non-functional approaches have been described in detail elsewhere (e.g. [13]), this paper provides only a brief overview of such languages here.

1.3.1 Using General Purpose Languages for Real-Time Programming

Historically, much embedded systems software/firmware was written for specific hardware using native assembler. Rapid increases in software and the need for productivity improvements mean that there has been a transition to the use of C/C++, and in some cases Java.Two extreme approaches to enforcing real-time properties in a language that is derived from a general-purpose design are exemplified by SPARK Ada [6] and the real-time specification for Java (RTSJ) [11]. SPARK Ada epitomises the idea of language design by elimination of unwanted behaviour from a general-purpose language, including concurrency. The remaining behaviour is guaranteed by strong formal models. In contrast, RTSJ provides specialised runtime and library support for real-time systems work, but makes no absolute performance guarantees. Thus, SPARK Ada provides a minimal, highly controlled environment for real-time programming emphasising *correctness by construction*,whilst Real-Time Java provides a much more expressible, but less controlled environment, without formal guarantees.

1.3.2 Domain-Specific Languages for Real-Time Programming

Process Algebra Derived Notations

Process algebras such as CSP, CCS, LOTOS and the π-calculus are formal notations designed to permit reasoning about complex systems of concurrent processes. They provide an elegant set of operators for developing concurrent systems, so allowing succinct expression of concurrent programs. Typical process

algebras use synchronous communication, support non-determinism, and allow choice, restriction of names and relabelling at the process level. Concurrency is usually modelled through interleaving processes. Process algebras provide a rich, tractable semantics, using *observation equivalence* to hide internal behaviours. This extensionalist approach contrasts with the intensionalist approach taken by Petri nets, where internal behaviour is important, and must consequently be exposed. Explicit notions of time have been incorporated into a number of process algebras, e.g. TCCS or Timed CSP. While process algebras are generally intended as formal notations to allow reasoning about concurrent specifications, there have also been some attempts to derive concrete programming notations from such bases. For example, LOTOS (Language of Temporally Ordered Specifications) is often used as a programming notation, and several timed extensions have been designed with the intention of dealing with real-time systems.

Finite-State Languages

Finite-state approaches are attractive when dealing with certain kinds of real-time system, since they allow a system to be defined by composing small, easily costed components. Such approaches often, however, prove problematic when constructing complex programs: typically the finite-state-machines derived for such systems will have a large number of states, which can be difficult for the programmer to manage; moreover relatively small extensions can cause exponential growth in the number of states. A number of extended finite-state languages have been proposed incorporating composition, communication, and data structures to give Turing-Complete notations. Many also incorporate quantitative notions of time. Three common examples are Estelle, an imperative language developed for OSI communications protocols; SDL, a language similar to Estelle, which has a graphical dialect used as a design tool; and TTM, a graphical notation similar to Petri nets, used to describe real-time discrete event processes.

Synchronous Dataflow Languages

In synchronous dataflow languages, every *action* (whether computation or communication) has a zero time duration. In practice this means that actions must complete before the arrival of the next event to be processed. Communication with the outside world occurs by reacting to external stimuli and by instantaneously emitting responses. Several languages have applied this model to real-time systems control. For example, Signal and Lustre are similar declarative notations, built around the notion of timed sequences of values. Esterel is an imperative notation that can be translated into finite state machines or hardware circuits, and Statecharts uses a visual notation, primarily for design. One obvious deficiency is the lack of expressiveness, notably the absence of recursion and higher-order combinators. Synchronous Kahn networks [15] incorporate higher-order functions and recursion, but lose strong guarantees of resource boundedness.

1.3.3 Functional Language Approaches

The main advantages of functional language approaches are compositionality, ease of reasoning and program structuring. Typical modern language designs, such as Standard ML or Haskell, incorporate *automatic memory management* which eliminates errors arising from poor manual memory management; *strong typing* which eliminates a large number of programming errors; *higher-order functions* which abstract over common patterns of computation; *polymorphism* which abstracts internal details of data structures; and *recursion* allows a number of algorithms, especially involving data structures, to be expressed in a more natural, and thus less error-prone fashion.

These language features improve productivity through raising the level of expressivity and program abstraction. However, they divorce the programmer from the ability to directly control program execution, and thus from a simple intuitive model of the program's time and space behaviour. Moreover, functional language implementations must bridge a larger gap between source language and concrete machine than is present with lower-level languages. This has historically led to a significant performance difference between functional languages and their imperative counterparts, and consequent doubt over the suitability of functional notations for real-time settings, where it is necessary to program within strong time and space bounds.

Compared with McDermid's criteria, the primary functional language designs thus meet the requirements for determinacy and correctness, but fail to deal effectively with asynchronicity, concurrency and bounded time and space. Concurrent extensions such as Concurrent ML [40] or Concurrent Haskell [36] add mechanisms for asynchronicity and concurrency, but likewise provide no bounded time or space guarantees. None of these notations provide mechanisms for periodic scheduling or interrupt handling, and all use a relatively low-level notion of thread and communication, with explicit message handling.

Soft Real-Time Functional Languages

The most widely used soft real-time functional language is the impure, strict language Erlang [4], a concurrent language with a similar design to Concurrent ML. Erlang has been used by Ericsson to construct a number of successful telecommunications applications in the telephony sector [10], including a real-time database, Mnesia [47]. Erlang is concurrent, with a lightweight notion of a process. Such processes are constructed using explicit spawn operations, with communication occurring through explicit send and receive operations to nominated processes. Finally, rather than exploiting static analysis order to ensure that hard dynamic resource bounds are achieved, the weakly typed Erlang relies exclusively on dynamic timeouts to meet soft real-time targets.

In contrast, Embedded Gofer is a strongly-typed purely functional programming language with a two-level structure, separating process and functional layers. It uses a monadic notation with explicit register access, processes and com-

munication, similar in kind to other explicitly concurrent programming notations. Unlike Erlang, Embedded Gofer is non-strict, raising questions about accurate static costing of programs (as opposed to dynamic *measurement* of typical runtime behaviour, which is not adequate to guarantee real-time behaviour). A similar approach has been taken by Fijma and Udink, who introduced special language constructs into Twentel to control a robot arm [19].

RT-FRP [45] builds on functional reactive programming embedded as a domain-specific language in Haskell to construct time and space bounded programs. RT-FRP is separated into a reactive part (comparable to a synchronous system) and a base part that must be guaranteed terminating and resource-bounded. It exploits tail-recursion across reactive components to encapsulate time and space resource usage within a single reactive component, and also supports integration across a series of reactive components. The work provides a formal operational semantics for resource consumption, which can be used to construct an automatic analysis to determine space and time bounds. Since RT-FRP is based on Haskell, of course, the underlying language implementation technology may affect timings and space usage through non-strict evaluation and non-real-time garbage collection. Consequently, in the current system, these bounds cannot be guaranteed. A different language substrate might, however, provide a better basis for these requirements. Finally, RT-FRP does not yet consider issues of periodic scheduling, and events are handled without regard to real-time concerns, such as dynamic memory allocation, making them unsuitable for low-level interrupt handling.

Finally, a number of reactive applications have been written in more conventional functional languages without recourse to even an incremental garbage collector or attempting to formally bound time or space behaviour. Examples include the impure Concurrent ML [40], and the purely functional Concurrent Haskell [36], Concurrent Clean [35] and Eden [12]. An interesting example of such work is the games engine and games written in Concurrent Clean [46].

Functional Languages for Mobility

Mobile languages focus on issues of security and portability rather than on time deadlines or absolute space usage. Mobile Haskell [38] is one functional notation that has explored the design space of mobile systems through exploiting a portable byte-code implementation that is capable of exporting and managing tasks across a distributed system.

A primary concern of mobile systems is to ensure that code that is generated at a remote site does not have unwanted local effects. These effects might be to access or alter local system state, so violating privacy, compromising security or damaging local data; or to either deliberately or accidentally overload local system resources. It follows that providing formally verifiable certificates of resource usage is important to mobile systems code. These certificates might include bounds on time and space usage and use a *proof-carrying code* approach.

This issue has been explored by the EU Framework V Mobile Resource Guarantees project in the shape of the Camelot and Grail notations [29]. Camelot is a

resource-aware functional programming language that can be compiled to a subset of JVM bytecodes; Grail is a functional abstraction over these bytecodes. This abstraction possesses a formal operational semantics that allows the construction of a program logic capable of capturing program behaviours such as time and space usage [5]. The objective of the work is to synthesise proofs of resource bounds in the Isabelle theorem prover, and to attach these proofs to mobile code in the form of more easily verifiable proof derivations. In this way the recipient of a piece of mobile code can cheaply and easily verify its resource requirements.

Functional Hardware Description Languages

In a slightly different context, functional *hardware description languages* [17, 27], also necessarily provide hard limits on time and space cost bounds. Like conventional finite-state notations, computation in such languages is necessarily restricted by the requirement to produce static hardware structures from the functional descriptions. The use of higher-order functions and recursion is thus restricted to forms that can be mapped to small finite structures. Examples of such notations include the Lava hardware description language for specifying FPGA circuits, which has been developed in association with XiLinx Corporation [17], the *functional derivation* approach, for deriving FPGA circuits from Haskell specifications [22], the Hawk hardware verification language [27], the Hydra system for logic circuit specification, and Mycroft and Sharp's statically allocated language for hardware description [34]. Like RT-FRP, most of these notations restrict recursion, if present, either to tail-recursion or to specific packaged, unfoldable recursive forms which can be used to generate repetitive circuits.

Functional Languages Imposing Syntactic Restrictions

Other than our own work [39, 44], we are aware of three main studies of formally bounded time and space behaviour in a functional setting [14, 25, 43]. All three approaches are based on restricted language constructs to ensure that bounds can be placed on time/space usage. In their recent proposal for Embedded ML, Hughes and Pareto [25] have combined the earlier *sized type system* [26] with the notion of *region types* [42] to give bounded space and termination for a first-order strict functional language [25]. Their language is restricted in a number of ways: most notably in not supporting higher-order functions, and in requiring the programmer to specify detailed memory usage through type specifications. The practicality of such a system is correspondingly reduced. Burstall[14] proposed the use of an extended *ind case* notation in a functional context, to define inductive cases from inductively defined data types. While *ind case* enables static confirmation of termination, Burstall's examples suggest that considerable ingenuity is required to recast terminating functions based on a laxer syntax. Turner's *elementary strong functional programming* [43] has similarly explored issues of guaranteed termination in a purely functional programming language. Turner's approach separates finite data structures such as tuples from potentially infinite

structures such as streams. This allows the definition of functions that are guaranteed to be primitive recursive, but at a cost in addtional programmer notation.

1.4 BOUNDING TIME AND SPACE USAGE

Garbage collection is both expensive and can introduce "embarrassing pauses" into a program execution. When the application is either soft- or hard- real-time, such pauses may be unacceptable. Two approaches have been taken to deal with this problem: real-time garbage collection techniques attempt to bound the cost of garbage collections to an acceptable level, thereby eliminating arbitrary pauses; while static analysis or compile-time garbage collection attempts to bound memory usage statically or eliminate garbage collection through memory reuse. The consequence is to transform dynamic heap-based memory allocation into either purely static memory allocation or some form of allocation that has predetermined extents, such as stack-based allocation. In this way, both allocation and collection costs can be determined in advance of program execution.

1.4.1 Real-Time Dynamic Memory Management

Effective management of dynamically allocated memory for a real-time system involves controlling the costs of both allocation and collection, ensuring that the system is *non-disruptive* in terms of meeting the application's real-time constraints. In memory constrained settings, it is also necessary to avoid wastage through fragmentation and other overheads. Developing an automatic memory management system for real-time systems represents a serious technical challenge. The Real-Time Specification of Java states, for example: "...the expert group believes, that no garbage collector algorithm or implementation is known ... which could be considered appropriate for all real-time systems" [11]. Many *non-disruptive* memory management systems require additional hardware support, which is not generally available, while others allocate memory only in fixed-size units, imposing potentially high memory overheads.

Most real-time memory management techniques use *Incremental garbage collectors*. Incremental copying techniques (e.g. [30]) achieve fast *allocation* but can have high memory overheads, and incur time overheads in the form of write- and/or read-barriers. Non-copying techniques such as those using incremental reference-counting [18] do not incur the overheads of copying, but may have poor memory utilisation due to external fragmentation (requiring an incremental compactor) and reference counts.

A number of such collectors have been proposed for use in functional language implementations. For example, Virding et al. have proposed an incremental collector for Erlang [2]; Wallace and Runciman have implemented an incremental collector for Embedded Gofer that has been used for undergraduate teaching at York University; and Cheadle et al. have implemented a similar incremental collector for the Glasgow Haskell compiler [16], though this has not yet been incorporated in the production release.

1.4.2 Static Analyses for Bounding Memory Usage

Compile-time garbage collection techniques attempt to eliminate some or all heap-based memory allocation through strong static means. One approach that has recently found favour is the use of region types [42]. Such types allow memory cells to be tagged with an allocation *region*, whose scope can be determined statically. When the region is no longer required, all memory associated with that region may be freed without invoking a garbage collector. In non-recursive contexts, the memory may be allocated statically and freed following the last use of any variable that is allocated in the region. In a recursive context, this heap-based allocation can be replaced by (possibly unbounded) stack-based allocation.

Hofmann's linearly-typed functional programming language LFPL [23] uses linear types to determine resource usage patterns. So-called *diamond* resource types are used to count constructors. First-order LFPL definitions can be computed in bounded space, even in the presence of general recursion. Hofmann has recently considered the extension of LFPL to higher-order functions with reference to *non size-increasing* recursive definitions on lists [24], where the size of all intermediate computations is bounded by the size of the inputs. Where definitions are restricted to primitive recursion only, this then guarantees polynomial size complexity. Unfortunately, for arbitrary higher-order functions, the cost of introducing closures means that an unbounded stack is required.

Building on earlier work on sized types [26, 39], we have developed an automatic analysis to *infer* the *upper bounds* on evaluation costs for a simple, but representative, functional language with parametric polymorphism, higher-order functions and recursion [44]. Our approach assigns finite costs to a non-trivial subset of primitive recursive definitions. It is *fully automatic* in producing cost equations without any user intervention, even in the form of type annotations, though obtaining closed-form solutions to the costs of recursive definitions currently requires the use of an external solver. The first-order subset of this work has been applied to our resource-bounded language Hume (Section 1.5.1).

1.4.3 Worst Case Execution Time Analysis.

Static analysis of *worst-case execution time* (WCET) in real-time systems is an essential part of the over-all response time and quality of service analysis [13, 37]. However, WCET analysis is a challenging issue, as the complexity of interaction between the software and hardware system components often results in very pessimistic WCET estimates. Recent work on WCET analysis for Java and C programs [7, 8] has employed a combination of analytical (in particular, probabilistic) and experimental (e.g. trace generation) techniques in order to reduce the degree of pessimism in WCET. However, the disadvantage of this approach is that it starts from a low-level code representation (Java byte-code or compiled machine code) which makes it difficult to capture and analyse the high-level program structure, and therefore to make predictions based on the programmer's intentions.

In an extension of work undertaken in EU project Daedalus, AbsInt have de-

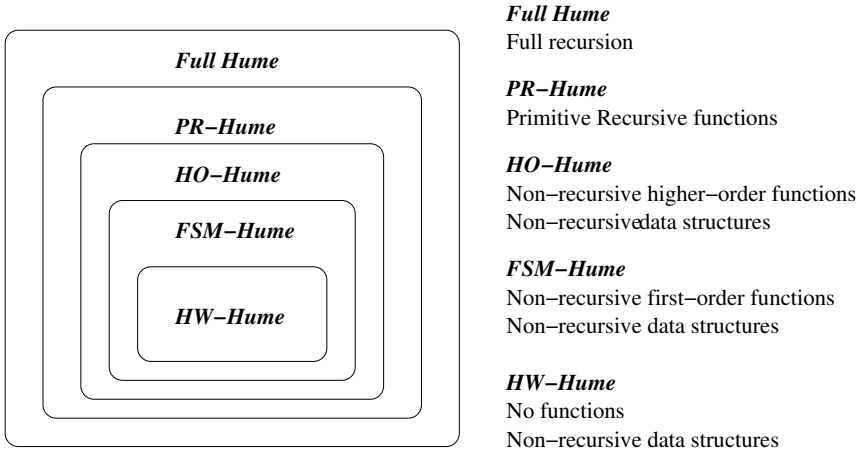

FIGURE 1.1. Hume Design Space

veloped accurate cost models for hardware instruction and cache behaviour for a number of architectures [28]. These models allow precise costing of execution times based on static analysis of machine code instructions. Compared with the *probabilistic* models that are commonly employed by WCET analyses, this approach allows vastly improved confidence in the quality of the analysis. Consequently, the reliability of real-time estimates can be raised dramatically for real architectures.

1.5 THE HUME LANGUAGE

The Hume language design attempts to maintain the essential properties and features required by the embedded systems domain (especially for transparent time and space costing) whilst incorporating as high a level of program abstraction as possible. We have designed Hume as a three-layer language [21]: an outer (static) declaration/metaprogramming layer, an intermediate coordination layer describing a static layout of dynamic processes ("boxes") and the associated devices, and an inner layer describing each process as a (dynamic) mapping from patterns to expressions. The inner layer is stateless and purely functional. Since boxes map bounded inputs to bounded outputs, real-time, bounded space responses to input requests can be ensured provided the functional expression layer can be determined to use finite space and execute in bounded time.

Rather than attempting to apply cost modelling and correctness proving technology to an existing language framework either directly or by altering the language to a greater or lesser extent (as with e.g. RTSj [11]), our approach is to design Hume in such a way that we are certain that formal models, proofs and the associated analyses can be constructed so as to ensure formally bounded time and space behaviour. We envisage a series of overlapping Hume language levels

application	predicted heap	actual heap	excess	predicted stack	actual stack	excess
pump controller	483	425	14.5%	166	162	2.5%
railway layout	1065	946	11%	310	310	0%
vehicle simulator	99408	98446	0.98%	319	298	6.5%

FIGURE 1.2. Heap and stack usage in words for FSM-Hume applications

as shown in Figure 1.1, where each level adds expressibility to the expression semantics, but either loses some desirable property or increases the technical difficulty of providing formal correctness/cost models.

1.5.1 Real Time and Space Behaviour of FSM-Hume Programs

We have applied our stack and heap analysis to a number of programs written using the FSM-Hume [33] language level[1]: a simple mine drainage pump controller; a model railway layout system with safety conditions; and a simulation of an autonomous vehicle controller [32]. Details of these applications can be found at http://www.hume-lang.org. Figure 1.2 shows results that are obtained from our analysis and prototype implementation. Note that any analysis (including one conducted by hand) must produce an over-estimate, to account for cases that by chance do not arise during the actual dynamic execution. With this caveat, we can see that the analysis is a good predictor of both stack and heap usage. Typically, we obtain better predictions of stack usage than heap. The memory used for the stack is also less than the heap usage.

We have ported the Hume implementation to the RTLinux real-time operating system. Our measurements [20] show that the total memory requirements of the pump application, including heap and stack overheads as calculated here, *RTLinux operating system code and data*, Hume runtime system code and data, and the abstract machine instructions amount to less than 62KB. RTLinux itself accounts for 34.4KB of this total. The results can be extrapolated to the other applications discussed here: the vehicle simulator would require much less than 512KB of dynamic memory, for example. Clearly, these results indicate both that tight dynamic memory bounds can be determined, and that these bounds are sufficiently small to allow implementation on typical modern embedded hardware.

To verify that our system can also meet real-time requirements, we have run the mine drainage control system continuously for a period of about 6 minutes under RTLinux on the same 1GHz Pentium III processor (effectively locking out all Linux processes during this period). At this point, the simulation has run to completion. Clock timings have been taken using the RTLinux system clock, which is accurate to the nanosecond level. The primary real-time constraint on the

[1] which admits first-order non-recursive functions in the functional expression layer, and a form of tail recursion in the coordination layer, analogously to RT-FRP.

mine drainage control system is that it must produce an alarm within 3ms if the methane level rises above some threshold. In fact, we have measured this delay to be approx. 150μs (20 times faster than required). Moreover, over the 6 minute time period, the *maximum delay* in servicing *any* input is approximately 2.2ms.

In order to demonstrate the robustness of the implementation within strong memory bounds, the vehicle simulation was run continuously under RT-Linux as a real-time program for a period of 36 hours using our calculated memory settings. The program ran without any memory accesses outside the allocated area, and without "growing" or "leaking" memory: essential requirements for real-time control applications. Total dynamic memory usage (*including code, runtime stack, and runtime libraries*) was 105340 words (412KB) of memory.

1.6 THE CHALLENGES

To summarise, while several functional notations have been proposed for soft real-time programming, Hume is the only language that we are aware of that has been shown to deal with hard real-time systems *in practice*, providing strong verifiable guarantees of space (and potentially) behaviour, and running under a true real-time operating system. To date this has been achieved only for the FSM-Hume level, however, which roughly corresponds to RT-FRP or synchronous dataflow designs plus first-order non-recursive functions. It is not clear whether formal analyses can be developed to deal with richer levels of Hume, including generalised forms of recursive definition and higher-order functions.

The primary issue facing functional languages as vehicles for programming real-time systems is whether they can meet the necessary strong time *and space* requirements, whilst simultaneously providing an effective means for programming with such behavioural concepts. Languages for real-time programming must incorporate notions of low-level behaviour including time, interrupts and scheduling. They must also accurately (formal and informal) reasoning about time and space usage from the high level source. This may be harder for functional languages to achieve because of the high level programming abstractions such as higher-order functions and polymorphic typing that make them attractive programming mechanisms. The *challenge* is to incorporate low level notions into the high level notation without compromising abstraction capability. This may involve a first-class treatment of real time and space and/or special language constructs. Such treatments are generally lacking in the literature.

At the same time, it is necessary to develop compilers for real-time functional languages that are both (adequately) high performance and highly verifiable. A number of languages (such as OCAML and SAC) demonstrated that strict functional languages can have extremely good time performance, and it is common to provide formal descriptions of functional abstract machine implementations in terms of formal or semi-formal transformation from the source level. The *challenge* is to combine the latter techniques with a mechanism such as Hofmann's *verifiable resource certificates* and to apply this to high performance functional language compilers. Moreover, optimising compilers must give proper attention

to space as well as time usage.

Cost analyses can help to provide information about time and space usage on an expression or program level. However, the current state of such analyses is that they require severe restrictions to the programming notations that can be used. For example, LFPL guarantees strong space bounds in a first-order context for programs that are linear [23]. Our own sized time analysis [44] will handle more general recursive, polymorphic programs, but the forms of recursion are restricted to simple inductions over natural numbers or linear data structures such as lists (in the form of primitive recursive cost equations) and there can be loss of quality in some important cases. Clearly more research is required if such analyses are to be exploited by Joe Functional Programmer.

Advances in compile-time garbage collection technologies such as regions [42] are welcome, but it does not seem possible to eliminate all dynamic memory allocation except in restricted settings such as FSM-Hume. Transforming heap allocations into stack allocations, as can happen with regions, increases memory residency, and the solution of reusing space through tail recursion is only a partial one. Thus, there is a need for good real-time garbage collectors. Unfortunately, non-disruptive garbage collectors tend to be accompanied by high memory overheads. The *challenge* is to devise a (hybrid?) memory management system that minimises memory overhead while providing real-time guarantees.

Finally, the majority of research into bounded time and space behaviour for functional languages has focused on strict notations. It is both much easier to provide strong formal cost models for strict languages, and to provide implementations that accurately reflect intuitions of time and space behaviour. Because evaluation is usually demand-based in a non-strict notation, it is an interesting and open question whether such demand can be predicted in such a way that it is possible to determine formal time or space bounds for the evaluation of a term. Analytical techniques will thus require good cost models to be combined with good resource usage models. Alternatively, it may be possible to produce a hybrid notation where real-time code is evaluated eagerly, and can thus exploit technology for strict notations, while non real-time code is evaluated lazily to provide good compositional capability. The *challenge* is to produce such a notation whose total space usage can be bounded in a sensible fashion.

1.7 CONCLUSION

Functional programming is potentially attractive for real-time systems because of its property of strong determinacy, and the promise of high levels of correctness. Moreover, higher-order functions and other mechanisms allow rapid program construction and restructuring (refactoring), leading to potential productivity advantages. However, issues relating to time and space management are key to the area, and until recently these have not been seriously considered by the community. Progress is being made on theoretical approaches that are geared towards bounding time and space usage, and many of these are couched in functional terms. There is, however, a gap between this and most existing practical work.

We have identified a number of challenges that are faced by functional language designers and implementors if real-time functional systems are to become truly feasible. Chief amongst these are serious consideration of time and space behaviour. It is necessary to raise time into the programming language in such a way that the real-time programmer can express real-time deadlines and constraints, and can guarantee that the program meets those constraints. It is also necessary to provide strong verifiable models of dynamic memory allocation that can be used to guarantee memory bounds, and to ensure that costs associated with automatic memory management do not adversely impact real-time deadlines.

REFERENCES

[1] P. Amey. Correctness by Construction: Better can also be Cheaper. *CrossTalk: the Journal of Defense Software Engineering*, pages 24–28, Mar. 2002.

[2] J. Armstrong. One Pass Real-Time Generational Mark-Sweep Garbage Collection. In *Proc. 1995 Intl. Workshop on Memory Management*, Kinross, Scotland, 1995.

[3] J. Armstrong. The Development of Erlang. In *Proc. 1997 ACM Intl. Conf. on Funct. Prog. (ICFP '97)*, pages 196–203, Amsterdam, The Netherlands, 1997.

[4] J. Armstrong, S. Virding, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.

[5] D. Aspinall, L. Beringer, M. Hofmann, and H.-W. Loidl. A Resource-Aware Program Logic for a JVM-like language. In *Proc. Implementation of Functional Languages (IFL '03)*. Springer-Verlag LNCS, 2004.

[6] J. Barnes. *High Integrity Ada: the Spark Approach*. Addison-Wesley, 1997.

[7] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *Proc. 12th Euromicro International Conference on Real-Time Systems*, Stockholm, June 2000.

[8] G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS 2002)*, Austin, TX. (USA), December 2002.

[9] G. Blair, L. Blair, H. Bowman, and A. Chetwynd. *Formal Specification of Distributed Multimedia Systems*. UCL Press, 1998.

[10] S. Blau and J. Rooth. AXD-301: a New Generation ATM Switching System. *Ericsson Review*, 1, 1998.

[11] G. Bollela and et al. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[12] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden Coordination Model for Distributed Memory Systems. In *Proc. High-Level Parallel Prog. Models and Supportive Envs. (HIPS)*, number 1123 in LNCS. Springer-Verlag, 1997.

[13] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages (Third Edition)*. Addison Wesley Longman, 2001.

[14] R. Burstall. Inductively Defined Functions in Functional Programming Languages. Technical Report ECS-LFCS-87-25, 1987.

[15] P. Caspi and M. Pouzet. Synchronous Kahn Networks. *ACM SIGPLAN Notices*, 31(6):226–238, 1996.

[16] A. Cheadle, A. Field, S. Marlow, S. P. Jones, and L. While. Non-Stop Haskell. In *Proc. 2000 ACM Intl. Conf. on Funct. Prog. (ICFP 2000)*, pages 257–267, 2000.

[17] K. Claessen and M. Sheeran. A Tutorial on Lava: a Hardware Description and Verification System. Aug. 2000.

[18] L. Deutsch and D. Bobrow. An Efficient Incremental Automatic Garbage Collector. *CACM*, 19(9):522–526, 1976.

[19] D. Fijma and R. Udink. A Case Study in Functional Real-Time Programming. Technical report, Dept. of Computer Science, Univ. of Twente, The Netherlands, 1991. Memoranda Informatica 91-62.

[20] K. Hammond. An Abstract Machine Implementation for Embedded Systems Applications in Hume. In *Submitted to IFL 2003*, 2003.

[21] 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. Springer-Verlag, 2003.

[22] J. Hawkins and A. Abdallah. Behavioural Synthesis of a Parallel Hardware JPEG Decoder from a Functitonal Specification. In *Proc. EuroPar 2002*, Aug. 2002.

[23] M. Hofmann. A Type System for Bounded Space and Functional In-place Update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[24] M. Hofmann. The strength of non size-increasing computation. In *Proc. 17th Annual IEEE Symposium on Logic in Computer Science*, pages 258–289, 2002.

[25] R. Hughes and L. Pareto. Recursion and Dynamic Data Structures in Bounded Space: Towards Embedded ML Programming. In *Proc. 1999 ACM Intl. Conf. on Functional Programming (ICFP '99)*, pages 70–81, 1999.

[26] R. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Proc. POPL'96 — 1996 ACM Symp. on Principles of Programming Languages*, St. Petersburg Beach, FL, Jan. 1996.

[27] J. L. J. Matthews and B. Cook. Microprocessor Specification in Hawk. In *Proc. International Conference on Computer Science*, 1998.

[28] D. Kästner. TDL: a Hardware Description Language for Retargetable Postpass Optimisations and Analyses. In *Proc. 2003 Intl. Conf. on Generative Programming and Component Engineering, – GPCE 2003, Erfurt, Germany*, pages 18–36. Springer-Verlag LNCS 2830, Sep. 2003.

[29] K. Mackenzie and N. Wolverson. Camelot and Grail: Compiling a Resource-Aware Functional Language for the Java Virtual Machine. In *this book*, 2004.

[30] B. Magnusson and R. Henriksson. Garbage Collection for Hard Real-Time Systems. Technical Report 95-153, Lund University, Sweden, 1995.

[31] J. McDermid. *Engineering Safety-Critical Systems*, pages 217–245. Cambridge University Press, 1996.

[32] G. Michaelson, K. Hammond, and J. Sérot. FSM-Hume: Programming Resource-Limited Systems using Bounded Automata. In *to appear in Proc. ACM Symp. on Applied Computing, Nicosia, Cyprus*, 2004.

[33] G. Michaelson, K. Hammond, and J. Sérot. The Finite State-ness of Finite State Hume. In *this book*, 2004.

[34] A. Mycroft and R. Sharp. A Statically Allocated Parallel Functional Language. *Automata, Languages and Programming*, pages 37–48, 2000.

[35] E. Nöcker, J. Smetsers, M. van Eekelen, and M. Plasmeijer. Concurrent Clean. In *Proc. Parallel Architectures and Languages Europe (PARLE91)*, number 505 in LNCS, pages 202–219. Springer-Verlag, 1991.

[36] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. POPL'96 — ACM Symp. on Principles of Programming Languages*, pages 295–308, Jan. 1996.

[37] P. Puschner and A. Burns. A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems*, 18(2/3):115–128, 2000.

[38] A. Rauber du Bois, P. Trinder, and H.-W. Loidl. Implementing Mobile Haskell. In *this book*, 2004.

[39] A. Rebón Portillo, K. Hammond, H.-W. Loidl, and P. Vasconcelos. A Sized Time System for a Parallel Functional Language (Revised). In *Proc. Implementation of Functional Langs.(IFL '02), Madrid, Spain*, number 2670 in Lecture Notes in Computer Science. Springer-Verlag, 2003.

[40] J. Reppy. CML: a Higher-Order Concurrent Language. In *Proc. 1991 ACM Conf. on Prog. Lang. Design and Impl. (PLDI '91)*, pages 293–305, June 1991.

[41] E. Schoitsch. Embedded Systems – Introduction. *ERCIM News*, 52:10–11, Jan. 2003.

[42] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 Feb. 1997.

[43] D. Turner. Elementary Strong Functional Programming. In *Proc. 1995 Symp. on Funct. Prog. Langs. in Education — FPLE '95*, LNCS. Springer-Verlag, Dec. 1995.

[44] P. Vasconcelos and K. Hammond. Inferring Costs for Recursive, Polymorphic and Higher-Order Functional Programs. In *Proc. Implementation of Functional Languages (IFL '03)*. Springer-Verlag LNCS, 2004.

[45] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *Intl. Conf. on Functional Programming (ICFP '01)*, Florence, Italy, September 2001. ACM.

[46] M. Wiering, P. Achten, and M. Plasmeijer. Using Clean for Platform Games. In *Proc. Implementation of Functional Languages (IFL '99)*, number 1868 in LNCS, pages 1–17. Springer-Verlag, 2000.

[47] C. Wikström and H. Nilsson. Mnesia — an industrial database with transactions, distribution and a logical query language. In *Proc. Intl. Symp. on Cooperative Database Systems for Advanced Applications*, 1996.