Trends in Functional Programming Volume 4

Edited by Stephen Gilmore

intellect

Trends in Functional Programming

Volume 4

Edited by
Stephen Gilmore

intellect™
Bristol, UK
Portland, OR, USA

First published in the UK in 2005 by
Intellect Books, PO Box 862, Bristol BS99 1DE, UK.
First published in the USA in 2005 by
Intellect Books, ISBS, 920 NE 58th Ave. Suite 300, Portland, Oregon 97213-3786, USA.

Copyright ©2005 Intellect Ltd.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission.

A catalogue record for this book is available from the British Library

Electronic ISBN 1-84150-915-9 / ISBN 1-84150-122-0
ISSN 1743-4505 (Print)

Printed and bound in Great Britain by Antony Rowe Ltd.

Contents

1 Is It Time for Real-Time Functional Programming? 1
- 1.1 Introduction . . . 1
- 1.2 What is Real-Time Programming? . . . 2
  - 1.2.1 The Importance of Real-Time Systems . . . 2
  - 1.2.2 Essential Properties of Real-Time Languages. . . . 2
- 1.3 Languages for Programming Real-Time Systems . . . 3
  - 1.3.1 Using General Purpose Languages for Real-Time Programming . . . 3
  - 1.3.2 Domain-Specific Languages for Real-Time Programming 4
  - 1.3.3 Functional Language Approaches . . . 5
- 1.4 Bounding Time and Space Usage . . . 7
  - 1.4.1 Real-Time Dynamic Memory Management . . . 7
  - 1.4.2 Static Analyses for Bounding Memory Usage . . . 7
  - 1.4.3 Worst Case Execution Time Analysis. . . . 8
  - 1.4.4 Syntactically Restricted Functional Languages . . . 9
- 1.5 Functional Languages for Related Problem Areas . . . 9
- 1.6 The Hume Language . . . 10
  - 1.6.1 Real Time and Space Behaviour of FSM-Hume Programs 12
- 1.7 The Challenges . . . 13
- 1.8 Conclusion . . . 14

2 FSM-Hume is Finite State 19
- 2.1 Introduction . . . 19
- 2.2 Single Box FSM-Hume Programs are Finite State . . . 22
- 2.3 Multi-Box FSM-Hume Programs are Finite State . . . 23
- 2.4 Example: Vehicle Simulation . . . 25
  - 2.4.1 Single-box FSM-Hume . . . 26
- 2.5 Conclusion . . . 28

3 Camelot and Grail: Resource-Aware Functional Programming for the JVM 29
- 3.1 Introduction . . . 29
- 3.2 Camelot . . . 30

3.2.1 Basic Features of Camelot 31
3.2.2 Diamonds and Resource Control 32
3.3 Grail 35
3.3.1 The Grail Type System 36
3.3.2 Compilation of Grail 36
3.4 Compiling Camelot to Grail 38
3.4.1 Representing Data 38
3.4.2 Compilation of Programs 39
3.4.3 Initial Transformations 40
3.4.4 Compilation of Expressions 41
3.5 Performance 41
3.6 Final Remarks 44

4 O'Camelot: Adding Objects to a Resource-Aware Functional Language 47
4.1 Introduction 47
4.2 Camelot 48
4.3 Extensions 49
4.4 Typing 53
4.5 Translation 55
4.6 Objects and Resource Types 57
4.7 Related Work 58
4.8 Conclusion 59

5 Static Single Information from a Functional Perspective 63
5.1 Introduction 63
5.2 Related Work 67
5.3 Static Single Information 68
5.4 Transformation 69
5.5 Optimistic versus Pessimistic 71
5.6 Converting Functional Programs Back to SSI 72
5.7 Motivation 73
5.8 Conclusions 74

6 Implementing Mobile Haskell 79
6.1 Introduction 79
6.2 Mobile Haskell 81
6.2.1 Communication Primitives 81
6.2.2 Discovering Resources 82
6.2.3 Remote Thread Creation 83
6.2.4 A Simple Example 83
6.3 Implementation Design 83
6.3.1 Introduction 83
6.3.2 Evaluating Expressions before Communication 84
6.3.3 Sharing Properties 85

6.3.4 MChannels . . . 86
6.4 The Implementation . . . 86
6.4.1 Packing Routines . . . 86
6.4.2 Communicating User Defined Types . . . 87
6.4.3 Evaluating Expressions . . . 88
6.4.4 Implementation of MChannels . . . 89
6.5 Initial Evaluation . . . 90
6.6 Related Work . . . 91
6.7 Conclusions and Future Work . . . 92

7 Testing Scheme Programming Assignments Automatically 95
7.1 Introduction . . . 95
7.2 WebAssign and AT(x) . . . 97
7.3 A Sample Session . . . 98
7.4 Structure of the AT(x) Framework . . . 100
7.4.1 Components of the AT(x) System . . . 100
7.4.2 Communication Interface of the Analysis Component . . . 101
7.4.3 Function and Implementation of the Interface Component 101
7.4.4 Global Security Issues . . . 103
7.5 The Core Analysis Component . . . 104
7.5.1 Requirements on the Analysis Components . . . 104
7.5.2 Analysis of Scheme Programs . . . 106
7.6 Implementation and Experiences . . . 107
7.7 Related Work . . . 108
7.8 Conclusions and Further Work . . . 109

8 Testing Reactive Systems with GAST 111
8.1 Introduction . . . 111
8.2 Overview of G∀ST . . . 112
8.2.1 Testing and Results . . . 113
8.2.2 Evaluating Test Results . . . 113
8.2.3 Logical Operators in G∀ST . . . 114
8.2.4 Automatic Generation of Test Values . . . 114
8.3 Specifying Reactive Systems in G∀ST . . . 115
8.3.1 Labelled Transition Systems . . . 116
8.3.2 Example: Conference Protocol . . . 117
8.3.3 Executing a Deterministic LTS . . . 118
8.3.4 The Implementation Under Test . . . 120
8.3.5 Testing the Conference Protocol . . . 120
8.3.6 Implementations with Other Types . . . 121
8.4 Better Test Data Generation from the LTS . . . 121
8.5 Functional and Nondeterministic Specifications . . . 123
8.6 Testing Nondeterministic Systems . . . 125
8.7 Related Work . . . 126
8.8 Conclusion . . . 127

PREFACE

This volume is the proceedings of the Fourth International Symposium on Trends in Functional Programming held in Edinburgh, on September 11th and 12th, 2003. For the first time this year the TFP symposium was co-located with the Implementation of Functional Languages workshop.

The Trends in Functional Programming series occupies a unique place in the spectrum of functional programming events because of its highly commendable policy of encouraging new speakers, particularly PhD students, to air their work to a receptive and friendly audience. By encouraging the next generation of functional programmers in this way the workshop helps to instill the understanding that functional programming is more than just syntax, semantics and type systems and nourishes the essence of the subject itself.

This year the papers from the workshop have addressed the research problems at the forefront of practical application of functional languages as in the papers on real-time functional programming in Hume from Kevin Hammond, Greg Michaelson and Jocelyn Serot and resource-bounded functional programming in Camelot from Kenneth MacKenzie and Nicholas Wolverson.

Functional programming languages are supported by sophisticated implementations. Two papers address this aspect of functional programming research, Jeremy Singer's paper on static single information and the paper on the implementation of Mobile Haskell from André Rauber Du Bois, Phil Trinder and Hans-Wolfgang Loidl.

For all of their virtues, functional programs are not automatically error-free so the book closes with two papers on testing functional programs from Manfred Widera and from Pieter Koopman and Rinus Plasmeijer.

I would like to thank the organisers of IFL, Abyd Al Zain, André Rauber Du Bois, June Maxwell, Greg Michaelson, Jan Henry Nyström and Phil Trinder for their work in organising the workshop registrations, the excursion, delegate packs, room bookings, audio-visuals and many other aspects of the event and for allowing the TFP meeting to make use of their industriousness in making all of this run smoothly.

My thanks also go to all of the authors for preparing their papers carefully using Hans-Wolfgang Loidl's LaTeX style file and to the referees for their thorough and rapid reviewing of the papers which were submitted.

The Trends in Functional Programming workshop gratefully acknowledges the support of the British Computer Society Formal Aspects of Computer Science special interest group.

Stephen Gilmore,
Edinburgh

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 all 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 (Sec. 1.6), 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 [15]. 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 and 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 per cent of all new processors are used in such systems [59].

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 [44].

- *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. [21]), this paper provides only a brief overview of such languages here. Berry [11] further considers the issue of whether to use general purpose or domain-specific languages for real-time programming.

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 [8] and the real-time specification for Java (RTSJ) [17]. 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.

A major issue for programming real-time embedded systems is memory management: it is essential both to bound memory usage and to control memory access time. When using general purpose languages, it is thus common to avoid recursive programming constructs (which may grow the stack in an "unrestricted" fashion) and also to avoid automatic dynamic memory allocation/collection. In Sec. 1.4 we describe some modern approaches that may allow the safe use of such constructs in a real-time embedded system.

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 one is 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 [20], an imperative language developed for OSI communications protocols; SDL [63], a language similar to Estelle, which has a graphical dialect used as a design tool; and TTM [49], a graphical notation, similar to Petri nets, used to describe real-time discrete event processes.

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 reaction to external stimuli and by instantaneous emission of responses. Because of their origin in the combination of control theory and computer science, synchronous notations have long been popular in the area of automatic control. Since they are equivalent to the zero-delay model of circuits, they have also more recently found employment in hardware design [12, 61].

Several languages have applied the synchronous model to real-time systems control. For example, Signal [28] and Lustre [50] are similar declarative notations, built around the notion of timed sequences of values. Esterel [18, 13, 14] is an imperative notation that can be translated into finite-state machines or hardware circuits, and Statecharts [31, 64] is a quasi-synchronous notation with a visual notation, which is primarily used for design, and which has been subsumed into UML [58]. One obvious deficiency of pure synchronous notations is the lack of expressive power, notably the absence of recursion and of higher-order combinators. Synchronous Kahn networks [39, 23] incorporate higher-order functions and recursion, but lose strong guarantees of resource boundedness. It is thus generally accepted [11] that pure synchronous languages are not powerful enough for complex systems programming and must interact with other languages and communication styles, in particular with asynchronous ones. There have consequently been some attempts to combine the two styles of programming, for example CRP [54] combines Esterel and CSP, and the Polis [7] hardware/software codesign system also employs Esterel in a mixed synchronous and asychronous setting.

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 [57] or Concurrent Haskell [51] 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 [16], including a real-time database, Mnesia [68]. 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 communication, 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 [27].

RT-FRP [66] 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 [57] and the purely functional Concurrent Haskell [51], Concurrent Clean [48] and Eden [19]. An interesting example of such work is the games engine and games written in Concurrent Clean [67].

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. Three 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; finally, language designs may be restricted so as to automatically bound time and/or memory usage.

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" [17]. 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. [43]) 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 [26] do not incur the overheads of copying, but may have poor memory utilisation owing 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 [24], 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 [60] that has

recently found favour is the use of *region types*. 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 [33, 35] uses linear types to determine resource usage patterns. A special resource type called "diamond" is used to count constructors. First-order LFPL definitions can be computed in linearly bounded space, even in the presence of general recursion. More recently, Hofmann and Jost have introduced [35] an automatic inference of these resource types and thus of heap-space consumption, using linear programming; at the same time, the linear typing discipline is relaxed to allow analysis of programs typable in a usage type system such as in [41, 6, 52].

Extensions of LFPL to higher-order functions have been studied in [34] where it was shown that such programs can be evaluated using dynamic programming in time $O(2^{p(n)})$ where n is the size of the input and p is a fixed polynomial. By a result of Cook this is equivalent to polynomial space plus an unbounded stack. With unrestricted use of higher-order functions, it remains an unsolved problem to turn this theoretical result into an efficient compilation scheme. If higher-order functions are used restrictively, as in the language C, then no closures are required and they can be "compiled away" without penalty.

Building on earlier work on sized types [37, 56], 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 [65]. 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 (Sec. 1.6.1).

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 [21, 53]. 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 [9, 10] 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 developed accurate cost models for hardware instruction and cache behaviour for a number of architectures [40]. 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.

1.4.4 Syntactically Restricted Functional Languages

Other than our own work [56, 65], we are aware of three main studies of formally bounded time and space behaviour in a functional setting [22, 36, 62]. 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 [36] have combined the earlier *sized type system* [37] with the notion of *region types* [60] to give bounded space and termination for a first-order strict functional language [36]. 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 [22] 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* [62] 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.

1.5 FUNCTIONAL LANGUAGES FOR RELATED PROBLEM AREAS

Functional Languages for Mobility

Mobile languages focus on issues of security and portability rather than on time deadlines or absolute space usage. Mobile Haskell [55] 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 [42]. 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* [25, 38] 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 [25], the *functional derivation* approach, for deriving FPGA circuits from Haskell specifications [32], the Hawk hardware verification language [38], the Hydra system for logic circuit specification, and Mycroft and Sharp's statically allocated language for hardware description [47]. 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.

1.6 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 [30]: 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 tech-

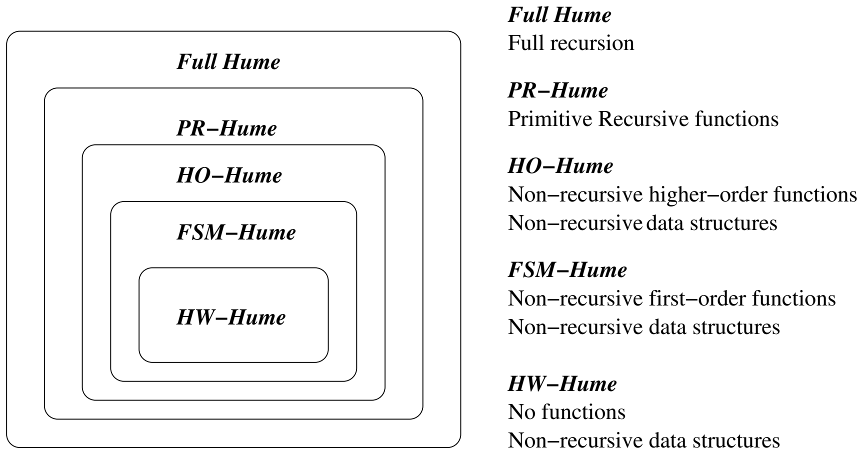

FIGURE 1.1 Hume Design Space

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

nology to an existing language framework either directly or by altering the language to a greater or lesser extent (as with e.g. RTSj [17]), 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 as shown in Fig. 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.

Hume thus meets McDermid's criteria as follows: *determinacy* is enforced at the language level, through a deterministic operational semantics; *bounded time/space* is ensured by the formal models and analyses for each Hume level; *asynchronous concurrency* is provided through concurrent boxes, with buffered communication and asynchronous pattern-matching rules; and *correctness* is assisted by the use of a purely functional expression layer and through the provision of formal language semantics. The design also incorporates periodic scheduling, interrupts and device polling.

1.6.1 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 [46] 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 [45]. Details of these applications can be found at http://www.hume-lang.org. Fig. 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 [29] 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 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 approximately 150μs (20 times faster than required). Moreover, over the six 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] 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.

1.7 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 support (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 [33]. Our own sized time analysis [65] 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 [60]

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.

1.8 CONCLUSION

Functional programming is potentially attractive for real-time systems because of its property of strong determinacy and the promise of easily constructing formal proofs 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] D. Aspinall and M. Hofmann. Another Type System for In-Place Update. In D. L. Metayer, editor, *Programming Languages and Systems (Proc. ESOP'02)*, volume Springer LNCS 2305, 2002.

[7] F. Balarin, M. Chiodo, A. Jurecska, H. Hsieh, A. L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Press, 1997.

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

[9] 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.

[10] 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.

[11] G. Berry. Real-time Programming: General Purpose or Special-Purpose Languages. *Information Processing*, 89:11–17, 1989.

[12] G. Berry. Esterel on Hardware. *Philosophical Transactions of the Royal Society of London*, 339:87–104, 1992.

[13] G. Berry and L. Cosserat. The Esterel Synchronous Programming Language and its Mathematical Semantics. In *Seminar on Concurrency*, volume 197 of *Lect. Notes in Computer Science*, pages 389–448. Springer Verlag, 1985.

[14] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Comp. Prog.*, 19(2):87–152, 1992.

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

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

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

[18] F. Boussinot and R. de Simone. The Esterel Language. *Proceedings of the IEEE*, 79(9):1293–1304, Sept. 1991.

[19] 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.

[20] S. Budkowski and P. Dembrinski. An Introduction to Estelle: a Specification Language for Distributed Systems. *Computer Networks and ISDN Systems*, 4:3–23, 1987.

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

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

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

[24] 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.

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

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

[27] 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.

[28] T. Gautier, P. L. Guernic, and L. Besnard. SIGNAL: A Declarative Language For Synchronous Programming of Real-Time Systems. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lect Notes in Computer Science*, pages 257–277. Springer-Verlag, 1987.

[29] K. Hammond. An Abstract Machine Implementation for Embedded Systems Applications in Hume. In preparation, 2004.

[30] 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.

[31] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

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

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

[34] M. Hofmann. The Strength of Non Size-Increasing Computation. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*. ACM Press, 2002.

[35] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL'03 — Symposium on Principles of Programming Languages*, New Orleans, LA, USA, Jan. 2003. ACM Press.

[36] 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.

[37] 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.

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

[39] G. Kahn. The Semantics of a Simple Language for Parallel Programming. *Information Processing*, 74:471–475, 1974.

[40] 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.

[41] N. Kobayashi and A. Igarashi. Resource Usage Analysis. In *POPL '02 — Principles of Programming Languages*, Portland, OR, Jan. 2002.

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

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

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

[45] 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.

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

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

[48] 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.

[49] J. Ostroff. A Logic for Real-Time Discrete Event Processes. *IEEE Control Magazine*, 10(2):95–102, 1990.

[50] N. H. P. Caspi, D. Pilaud and J. Place. Lustre: a Declarative Language for Programming Synchronous Systems. In *Proc. 14th ACM Symposium on Principles of Programming Languages (POPL '87), München, Germany*, 1987.

[51] 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.

[52] S. Peyton-Jones and K. Wansbrough. Simple Usage Polymorphism. In *Proc. 3rd ACM SIGPLAN Workshop on Types in Compilation, Montreal*, September 2000.

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

[54] S. Ramesh, G. Berry, and R. K. Shyamasundar. Communicating Reactive Processes. In *Proc. 20th ACM Conf. on Principles of Prog. Langs. (POPL '93)*, 1993.

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

[56] 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.

[57] 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.

[58] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

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

[60] M. Tofte and J.-P. Talpin. Region-Based Memory Management. *Information and Computation*, 132(2):109–176, 1 Feb. 1997.

[61] H. Touati and G. Berry. Optimised Controller Synthesis using Esterel. In *Proc. Intl. Workshop on Logic Synthesis (IWLS 93), Lake Tahoe*, 1993.

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

[63] I. T. Union. [Z.100] Recommendation Z.100 (11/99) – Specification and description language (SDL). 1999.

[64] D. Varro. A Formal Semantics of UML Statecharts by Model Transition Systems. In *Proc. ICGT 2002: International Conference on Graph Transformation*, 2002.

[65] 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.

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

[67] 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.

[68] 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.

Chapter 2

FSM-Hume is Finite State

Greg Michaelson,[1] Kevin Hammond[2] and Jocelyn Serot[3]

Abstract Hume is a domain-specific programming language targeting resource-bounded computations. It is based on generalised concurrent bounded automata, controlled by transitions characterised by pattern matching on inputs and recursive function generation of outputs. Here we discuss the design of FSM-Hume, a strict finite state subset of Hume, and suggest that it is indeed classically finite state.

2.1 INTRODUCTION

We would like to be able to prove automatically the correctness, equivalence, termination, space use and complexity of arbitrary programs but these properties are all undecidable for Turing-complete (TC) languages [1]. Some decidability may be achieved by restricting the types and constructs in a language. Languages based on primitive recursion, such as Turner's elementary strong functional programming [6] or Burstall's inductively defined functions [2], seem unwieldy and to lack clear programming methodologies. Languages based on finite state automata (FSA), such as Promela with the related Spin model checker [4], have proved much more successful, but of relatively limited application and with vast state spaces, constraining verification of substantial programs.

Hume [3] is based on a generalisation of standard FSA transition notation to encompass a full TC language. Concurrent processing is based on explicit multiple communicating FSA, called boxes. Within Hume, an explicit distinction is made between the coordination language, which describes external properties and configurations of boxes, and the expression language, which describes input/output transitions within boxes. Finally, in full Hume, both sub-languages

[1]School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton, Scotland, EH14 4AS, `greg@macs.hw.ac.uk`

[2]School of Computer Science, University of St Andrews, North Haugh, St Andrews, Scotland, KY16 9AJ, `kh@dcs.st-and.ac.uk`

[3]LASMEA, Blaise Pascal University, Les Cezeaux, F-63177 Aubiere cedex, France, `Jocelyn.Serot@lasmea.univ-bpclermont.fr`

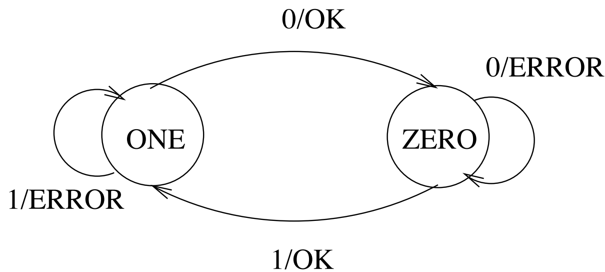

FIGURE 2.1. Mealy machine for alternating `1`s and `0`s

share a rich, polymorphic type system. These design decisions enable us to identify layers of language in Hume, with different decidable properties, which may be supported by high-level cost models [5].

A FSA with output (Mealy machine) is usually characterised by transition quadruplets of the form: $(old\ state, input) \rightarrow (new\ state, output)$
where *old state*, *input*, *new state* and *output* are finite sets, for example, the Mealy machine which checks that a binary sequence has alternating `1`s and `0`s, shown in Fig. 2.1, has transitions:

```
(ZERO,0) -> (ZERO,ERROR)
(ZERO,1) -> (ONE,OK)
(ONE,0) -> (ZERO,OK)
(ONE,1) -> (ONE,ERROR)
```

However, both the diagrammatic and state transition characterisations are misleading. First of all, it is implicit that a FSA cycles indefinitely, communicating with an external environment to consume single input symbols and generating single output symbols. Secondly, it is implicit that a FSA retains its state in between cycles. The external input/output links and state retention are made explicit for the above example in Fig. 2.2.

In general, for one FSA it need not be specified where the input comes from or where the output goes to: both could be linked to arbitrary sources and sinks, including to other FSA. Similarly, in principle, the old and new state need not be a direct feedback link but could again come via arbitrary sources and sinks, including other FSA.

The state and I/O symbol sets for a FSA must be finite but they may also be very big. Given a large enough set that maps to integers, then complex data structures may be encoded using either Gödel numbers within the set, or, more familiarly, structured ASCII sequences whose concatenated bit values are integers within the set.

Noting that the left and right hand sides of traditional transitions are like two-element tuples, we generalise them to: $pattern \rightarrow expression$. Here the left hand side $pattern$ is composed of variables, constants and structures. Note the wildcard pattern `*` which ignores the corresponding inputs without consuming it. Similarly,

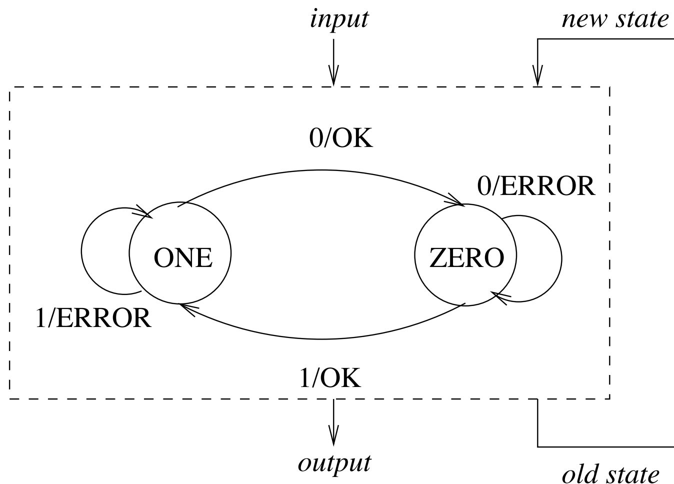

FIGURE 2.2. Mealy machine with explicit I/O and state

the right hand side *expression* may involve the components of the *pattern*, in particular the variables it introduces.

Thus, we generalise a FSA to a box with multiple input and output wires, where the state is no longer necessarily distinguishable from the input or output. Operationally, a box cycles repeatedly, trying to match transition *pattern*s against the current values on the input wires, treated as a single top-level tuple value. For a match to succeed, constants and constructors must appear in the same positions in the pattern and input value. Variables in the *pattern* are then instantiated to corresponding components of the input value. After a successful match, the output wires are instantiated from the tuple of values generated by the transition's right hand side.

For example, we can write the above Mealy machine in Hume as:

```
type BIT = int 1;
data STATE = ZERO | ONE;
stream Input from "std_in";
stream Output to "std_out";

box Bits
in (oldstate::STATE,input::BIT)
out (newstate::STATE,output::string)
match
 (ZERO,0) -> (ZERO,"ERROR\n") |
 (ZERO,1) -> (ONE,"OK\n") |
 (ONE,0) -> (ZERO,"OK\n") |
 (ONE,1) -> (ONE,"ERROR\n");
```

```
wire Bits (Bits.newstate initially ZERO,Input)
          (Bits.oldstate,Output);
```

Full Hume has constructs found in a contemporary polymorphic functional language, including recursive, unbounded, user-defined types. Finite State Machine Hume (FSM-Hume) is the Hume layer with finite types on wires and only simple operations, such as boolean and arithmetic, in transition expressions.

It might be thought that allowing operations whose state space is larger than the input space, such as multiplication, would transcend finite state-ness. However, for fixed precision numbers, it is possible to build a FSA that will carry out multiplication for values whose multiples do not exceed the largest allowed value, for example by encoding the appropriate look up table.

It might also be thought that Hume suffers from the same problems as other FSA-based languages, in particular state space explosion for practical verification of realistic programs. However, given appropriate transformation techniques, it should be possible to convert multiple boxes employing an impoverished expression language to fewer boxes using a richer expression language. Gross properties of box internals would still have to be established, using, say, automated theorem proving, but the state space of the overall box system would have been reduced. The balance between model checking and theorem proving in establishing properties of Hume programs is an interesting avenue of research which is not discussed further here.

A more serious concern is to clarify in what sense a multi-box Hume program is actually still a FSA, given the presence of multiple inputs and outputs, and the withering away of the state. We first discuss the status of a single box program and then explore multi-box programs.

Note that the following sections provide an informal framework for possible formalisation and are intended to convey conviction rather than establish correctness.

2.2 SINGLE BOX FSM-HUME PROGRAMS ARE FINITE STATE

Consider a Hume box with multiple inputs and outputs, and no distinguished state. As noted above, multiple values from finite domains, represented as a fixed width tuple, can be encoded as a single symbol, given a large enough space of symbols. Thus a box with multiple inputs or outputs may be treated as if it had just one input and output, each bearing a tuple value.

A multi-state FSA may be converted to a single state FSA as follows. The state symbol in each transition is combined with the input/output symbols in tuples. Each transition is then extended with a new single state value, in the state position on the left and right hand sides. In general:

$$(old\ state, input) \to (new\ state, output) \Longrightarrow$$
$$(single\ state, (old\ state, input)) \to (single\ state, (new\ state, output))$$

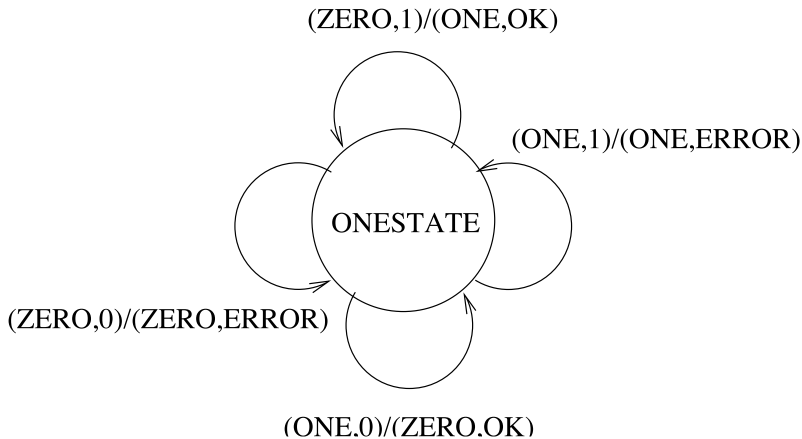

FIGURE 2.3. Single state Mealy machine for alternating `1`s and `0`s

For example, the Mealy machine above might be changed as shown in Fig. 2.3, with transitions:

```
(ONESTATE,(ZERO,0)) -> (ONESTATE,(ZERO,ERROR))
(ONESTATE,(ZERO,1)) -> (ONESTATE,(ONE,OK))
(ONESTATE,(ONE,0)) -> (ONESTATE,(ZERO,OK))
(ONESTATE,(ONE,1)) -> (ONESTATE,(ONE,ERROR))
```

Using this technique, a Hume box with multiple inputs and outputs, and no distinguished state, may be converted directly to a single state FSA with single composite input and output tuples, provided it has no variables in transition *pattern*s. A variable in a *pattern* corresponds to successfully matching any value in the domain for the variable's type. Thus, to fully convert a Hume transition with variables to pure FSA form, it must be replaced by multiple copies, with one copy for each combination of variable type domain values.

2.3 MULTI-BOX FSM-HUME PROGRAMS ARE FINITE STATE

We also need to convince ourselves that a multi-box FSM-Hume program is still finite state. If such a program may be converted into a single box FSM-Hume program then that program is finite state by the preceding argument.

Hume box scheduling is well defined as sequential, round robin where each box takes in it turns to execute once, in fixed sequence. For a multi-box program, we combine the box transitions and introduce an explicit state value to ensure sequentiality. Essentially, each transition for the combined box will correspond to a transition of one of the separate boxes, augmented with additional left hand side *pattern*s and right hand side *expression*s to circulate the wire values for all the other boxes without changing them.

In general, a successful transition for any one box must be able to transmit all possible wire values for the other boxes: any one box must be able to succeed if its inputs are matched successfully, regardless of the values on the wires for the other boxes. We employ Hume variables to generalise arbitrary input values, noting that they may in turn be replaced by all possible values of the corresponding types for pure FSAness, at the cost of a huge explosion in code size.

Suppose there are N boxes and box i has I_i inputs ($in_{i1}...in_{I_i}$) and O_i outputs ($out_{i1}...out_{O_i}$).

For each box, we construct a top level pattern template:

P_i: $var_{i1},var_{i2}...var_{iI_i}$

with a unique variable for each input. We also construct a top level expression template:

E_i: $var'_{i1},var'_{i2}...var'_{iO_i}$

where var'_{ij} is the new variable corresponding to the box input to which output out_{ij} is connected.

We then form a top level template for the transitions of the composite box by concatenating together the box pattern templates on the left and expression templates on the right:

$(P_1,P_2...P_N) \rightarrow (E_1,E_2...E_N)$

This template accepts arbitrary inputs and sends them to the appropriate outputs unchanged.

Suppose box i has T_i transitions, where the kth is: t_{ik}: $patt_{ik} \rightarrow exp_{ik}$
Then for each transition of box i, t_{ik}, we make a copy of the composite box's top level template, replace the pattern template P_i with the pattern $patt_{ik}$ and replace the expression template E_i with the expression exp_{ik}:

$(P_1...patt_{ik}...P_N) \rightarrow (E_1...exp_{ik}...E_N)$

Where the expression is a condition, the right hand side of the template must be pushed through to the condition options. Similarly, where the expression is a definition, the right hand side of the template must be pushed through to the result expression.

After this stage, where any remaining pattern template has a variable which has been replaced by an expression on the right hand side, then that variable should be replaced by the "ignore" pattern `*`: there should not be an input value present for that variable because a new value has been output for it. Similarly, where any expression template has a variable that was replaced in a pattern template, then that variable must be replaced by the "no output" operator `*`: the input has been consumed and cannot be re-circulated.

We are then left with common variables between left and right hand sides which consume inputs and reproduce them as outputs, to act as the inputs again on the next cycle. The effect is as if the corresponding wires had been ignored.

Thus, all variables on the left/right of a transition which are not in that transition's replacement pattern may be replaced by the "ignore" pattern/"no output" `*`.

Next, we introduce an explicit state which changes on each transition. We precede each composite pattern with the number of the corresponding box and each composite expression with the number of the next box:

$$(i,*,...,*,patt_{ik},*,...,*) \rightarrow (i+1,*,...*,exp_{ik},*,...,*)$$

or, for the last box, with the number of the first box.

Finally, we combine the wiring for each box, again adding a new feedback wire for the new explicit state.

The effect is two-fold. From a Hume perspective, we have constructed a single box which emulates multi-box scheduling. From a FSA perspective, we can easily convert the composite box into a FSA, with an explicit state, and composite input and output, using the technique described above.

2.4 EXAMPLE: VEHICLE SIMULATION

We now illustrate this transformation with reference to the simulation of a simple autonomous vehicle, which tries to follow a white line by repeatedly analysing a camera image consisting of one row of bits from a two-dimensional bit-map scene, effectively a map of the terrain the vehicle is traversing. The vehicle has a location consisting of its Cartesian coordinates in terrain space and its angle of orientation relative to the horizontal. The vehicle sends its current location to the environment. If the vehicle has not "bumped" into the edge of the terrain then the environment returns an image corresponding to the vehicle's position. The vehicle then sends the image to the control which calculates a new orientation to try to bring the white line back into the centre of the image. Finally, the vehicle changes its position and requests the next image from the environment. The vehicle also sends monitoring information to standard output:

```
box env in (loc::location) out (v::image,b::bool)
 match loc -> if within_scene loc
              then (lookat loc, false)
              else (null_image, true);

wire env (vehicle.loc initially init_loc)
         (vehicle.v, vehicle.b);

box vehicle
 in (v::image,b::bool,ploc::location,c::real)
 out (loc::location,m::monitor,
      loc'::location,v'::image)
 match
  (v, false, pl, c) ->
   let nl = move pl c
```

```
   in (nl, (v,pl,false,c,'\n'), nl, v)
| (v, true, pl, c) ->
   (init_loc, (v,pl,true,c,'\n'),
    init_loc, lookat init_loc);

wire vehicle
     (env.v,env.b,vehicle.loc' initially init_loc,
      control.da initially 0.0)
     (env.loc,std_out,vehicle.ploc,control.v);

box control in (v::image) out (da::real)
 match
   <<_,_,_,_,_,_,_,1,_,_,_,_,_,_,_>> -> 0.0
 ...
 | _ -> 0.0 ;

wire control (vehicle.v') (vehicle.c);
```

The simulation runs in real time and the vehicle never deviates more than a few bits to either side of the line.

2.4.1 Single-box FSM-Hume

First we construct the pattern templates and then the expression templates using the variable names from the pattern templates. We adopt the convention of naming template variables by preceding each input wire's name with a letter to denote its box name:

control pattern: `c_v`; env pattern: `e_loc`;
vehicle pattern: `v_v,v_b,v_ploc,v_c`
control expression: `v_c`; env expression: `v_v,v_b`;
vehicle expression: `e_loc,o,v_ploc,c_v`

i.e. the `control` output is wired to the `vehicle` input `c`; the env output is wired to the `vehicle` inputs `v` and `b`; etc.

The overall transition template is:

```
c_v,e_loc,v_v,v_b,v_ploc,v_c ->
 v_c,v_v,v_b,e_loc,o,v_ploc,c_v
```

Consider the first transition for the `control`. In the template, we replace `c_v` on the left with the transition pattern, `v_c` on the right with the transition expression and all other variables with `*`.

Consider the transition for the `env`. In the template, we replace `e_loc` on the left with the pattern. The transition expression is a conditional expression so we leave the condition in place, replace the option expressions with the template right hand side and insert the components expressions in place of the corresponding template variables `v_v` and `v_b`. Again, all other variables are replaced by `*`.

Consider the first transition for the `vehicle`. In the template, we replace `v_v`, `v_b`, `v_ploc` and `v_c` with the pattern components. There is a local definition on the right so we leave the declaration part in place, replace the expression with the template right hand side and insert the components of the expression in place of the corresponding template variables `e_loc`, `o`, `v_ploc` and `c_v`. Again, all other variables are replaced by `*`.

Numbering the boxes `control`/1, `env`/2 and `vehicle`/3, we add state patterns and expressions to each transition:

```
box vehicle
in (s::integer,c_v::image,e_loc::location,
    v_v::image,v_b::bool,v_ploc::location,v_c::command)
out (s'::integer,c_da::real,e_v::image,e_b::bool,
     v_loc::location, v_m::monitor,v_loc'::location,
     v_v'::image)
match
 (1,<<_,_,_,_,_,_,_,1,_,_,_,_,_,_,_>>,*,*,*,*,*) ->
  (2,0.0,*,*,*,*,*,*) |
 ...

 (2,*,loc,*,*,*,*) ->
      if within_scene loc
      then (3,*,lookat loc, false,*,*,*,*)
      else (3,*,null_image, true,*,*,*,*) |

 (3,*,*,v, false, pl, c) ->
  let nl = move pl c
  in (1,*,*,*,nl, (v,pl,false,c,'\n'), nl, v) |
 ...
```

Finally, we amalgamate the box wires and add appropriate wiring for the state, to start with the `env` box in state 2:

```
wire vehicle
 (vehicle.s' initially 2,vehicle.v_v',
  vehicle.v_loc initially init_loc,
  ...
  vehicle.c_da initially 0.0)
 (vehicle.s,vehicle.v_c,vehicle.v_v,vehicle.v_b,
  vehicle.e_loc, output,vehicle.v_ploc,vehicle.c_v);
```

The single box version of the vehicle simulation gives the same behaviour as the multi-box version, on the full Hume interpreter and on the HAM. It is also substantially faster and requires substantially less space.

2.5 CONCLUSION

We have explored the specific properties of the Hume finite state subset FSM-Hume to demonstrate informally that it is indeed finite state. In so doing, we derived a transformation to convert multi-box FSM-Hume programs to a single box and applied it to the simulation of a simple line following vehicle. We now plan to formalise and prove the transformation.

The application of the transformation to the vehicle simulation was performed by hand. We also plan to automate the transformation and to perform further experimentation to determine whether this transformation is a useful optimisation for general FSM-Hume programs.

This work has been partly supported by UK EPSRC grant GR/R 70545/01 and by a French CNRS grant.

REFERENCES

[1] W. S. Brainerd and L. H. Landweber. *Theory of Computation*. Wiley, 1974.

[2] R. Burstall. Inductively Defi ned Functions in Functional Programming Langauges. Technical Report ECS-LFCS-87-25, LFCS, University of Edinburgh, April 1987.

[3] K. Hammond and G. Michaelson. Hume: a Domain-Specifi c Language for Real-Time Embedded Systems. In *Proc. 2003 Intl. Conf. on Generative Programming and Component Engineering, – GPCE 2003, Erfurt, Germany*, LNCS, pages 37–56. Springer-Verlag, 2003.

[4] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.

[5] G. Michaelson, K. Hammond, and J. Serot. FSM-Hume: Programming Resource Limited Systems with Bounded Automata. In *Proc. ACM Symposium on Applied Computing (SAC'04), Cyprus, March 2004*. ACM Press, to appear.

[6] D. Turner. Elementary Strong Functional Programming. In *Proc. 1st Int. Symp. on Functional programming Languages in Education,Holland*, volume 1022 of *LNCS*. Springer, december 1995.

Chapter 3

Camelot and Grail: Resource-Aware Functional Programming for the JVM

K. MacKenzie[1] and N.Wolverson[1]

Abstract We describe the functional language Camelot, which is a language of the ML family with extensions for explicit management of heap storage, and the intermediate language Grail, which is a functional form of JVM bytecode. A scheme for transforming Camelot into Grail is described. We also give some figures for execution times which show that Camelot programs perform reasonably well when compared with Java equivalents.

3.1 INTRODUCTION

The Mobile Resource Guarantees (MRG) project [15] aims to develop a Proof Carrying Code (PCC) [16] framework to endow mobile computer programs with guarantees of resource bounds. Typical resources are time, heap space, system calls, and stack size. Our goal is to provide a resource-safe programming language to be used for writing mobile code. This language, which is called Camelot, is a high-level functional language which is compiled into JVM bytecode. The class files produced by the compiler will be equipped with a proof that the programs obey specified resource constraints and can then be transmitted across a network in the usual way. The consumer of the mobile code can then independently verify the resource constraints by checking the proof attached to the code; if verification is successful then execution can proceed as normal. This technique provides an unforgeable guarantee that the claimed resource limits will not be exceeded.

[1]Laboratory for the Foundations of Computer Science, School of Informatics, The University of Edinburgh, Mayfield Road, Edinburgh EH9 3JZ, Scotland, UK; Email: `kwxm@inf.ed.ac.uk`, `N.Wolverson@sms.ed.ac.uk`

In this paper we will describe Camelot and its translation to JVM bytecode. Camelot is similar to a subset of O'Caml, the main novelty lying in extensions for performing in-place modifications to heap-allocated data-structures. These features are similar to those described in by Hofmann in [6] but include some extra extensions for freelist management. To retain a purely functional semantics for the language in the presence of these extensions a linear type system can be employed: in the present implementation, linearity can be enforced via a compiler switch. We are in the process of enhancing the compiler by the addition of other, less restrictive type systems which still allow safe in-place modifications. More details will be given below.

Crucial design choices for the compilation are transparency and an exact specification of the compilation process. The former ensures that the compilation does not modify the resource consumption in an unpredictable way. The latter provides a formal basis for using resource information inferred for the high-level language in proofs on the intermediate language.

3.2 CAMELOT

Camelot is a strongly typed language of the ML family with features added to enable close control of heap usage. The syntax of Camelot (which is similar to a subset of the syntax of the O'Caml language [17]) is given below. The terms *tycon*, *cname*, *fname* and *var* refer to *type constructors*, *constructor names*, *function names* and *variable names* respectively: all of these are names in SML style. Constructor names must begin with an upper-case letter, whereas all other identifiers begin with a lower-case letter. The term *tyvar* refers to a *type variable*, which is a name beginning with a single quote. Literal constants (*const* below) are similar to those in O'Caml. Optional items are enclosed in angular parentheses.

$$
\begin{array}{rcl}
\textit{program} & ::= & \langle \textit{typedecseq} \rangle\ \langle \textit{valdecseq} \rangle\ \langle \textit{funimpseq} \rangle \\
\textit{typedecseq} & ::= & \textit{typedec}\ \langle \textit{typedecseq} \rangle \\
\textit{typedec} & ::= & \texttt{type}\ \langle \texttt{(}\textit{tyvar}_1 \ldots \textit{tyvar}_n\texttt{)} \rangle\ \textit{tycon}\ \texttt{=}\ \textit{conbind} \\
\textit{conbind} & ::= & \textit{cname}\ \langle \texttt{of}\ \textit{ty}_1 * \ldots * \textit{ty}_n \rangle\ \langle\ \texttt{|}\ \textit{conbind} \rangle \\
 & & \mid\ \texttt{!}\textit{cname}\ \langle\ \texttt{|}\ \textit{conbind} \rangle \\
\textit{ty} & ::= & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{float} \mid \texttt{string} \mid \textit{tyvar} \\
 & & \mid \textit{ty}\ \texttt{array} \mid \textit{tyseq}\ \textit{tycon} \mid \textit{ty}_n\ \texttt{->}\ \ldots\ \texttt{->}\ \textit{ty}_1\ \texttt{->}\ \textit{ty} \\
\textit{valdecseq} & ::= & \textit{valdec}\ \langle \textit{valdecseq} \rangle \\
\textit{valdec} & ::= & \texttt{val}\ \textit{var}\texttt{:}\ \textit{ty} \mid \texttt{val}\ \textit{fname}\texttt{:}\ \textit{ty} \\
\textit{funimpseq} & ::= & \textit{funimp}\ \langle \textit{funimpseq} \rangle \\
\textit{funimp} & ::= & \texttt{let}\ \langle \texttt{rec} \rangle\ \textit{fundecseq} \\
\textit{fundecseq} & ::= & \textit{fundec}\ \langle \texttt{and}\ \textit{fundecseq} \rangle \\
\textit{fundec} & ::= & \textit{fname}\ \textit{varseq}\ \texttt{=}\ \textit{expr}
\end{array}
$$

expr ::= *const* | *var* | *uop expr* | *expr op expr* | *fname* $expr_1 \ldots expr_n$
  | *cname* `(`$expr_1, \ldots, expr_n$`)` | *cname* `(`$expr_1, \ldots, expr_n$`)@`*var*
  | `let` *pat* `=` *expr* `in` *expr* | `if` *expr* `then` *expr* `else` *expr*
  | `match` *expr* `with` *match* | `free` *var* | `(`*expr*`)` | `begin` *expr* `end`

match ::= *mrule* ⟨ `|` *match*⟩

mrule ::= *con*⟨`(`$pat_1, \ldots, pat_n$`)`⟩ `->` *expr*
  | *con*⟨`(`$pat_1, \ldots, pat_n$`)`⟩`@`*pat* `->` *expr*

pat ::= *var* | _

uop ::= `-` | `-.` | `not`

op ::= *arithop* | *cmp* | `^` | `&&` | `||`

arithop ::= `+` | `-` | `*` | `/` | `mod` | `+.` | `-.` | `*.` | `/.`

cmp ::= `=` | `<` | `<=` | `>=` | `>` | `=.` | `<.` | `<=.` | `>=.` | `>.`

There are a number of built-in operators: the operators `+`,`-`, ... `>` apply to integer values, whereas `+.`,`-.`, ... apply to floating-point values. The boolean expression e_1 `&&` e_2 is an abbreviation for `if` e_1 `then` e_2 `else false`; similarly e_1 `||` e_2 represents `if` e_1 `then true else` e_2. The remaining binary operator is `^`, which performs string concatenation. There are also three unary negation operators.

In addition there are a number of predefined functions such as `print_int`, `print_int_newline`, and `int_of_float`, whose names should explain themselves. The `same_string` function is used to compare strings for equality. There are functions for handling arrays, but we will not use these here. Camelot also includes a built-in polymorphic `list` type. In order to execute a program the user must include a function `start: string list -> unit` ; when the class file is executed the `start` function will be executed with an argument consisting of a list containing the command-line arguments to the program.

Note that in some contexts the symbol _ can be used instead of a variable name. This feature can be used to discard unwanted values such as unit values returned by print statements.

3.2.1 Basic Features of Camelot

The core of Camelot is a standard polymorphic ML-type functional language. One can define datatypes in the normal way:

```
type intlist = Nil | Cons of int * intlist
type 'a polylist = NIL | CONS of 'a * 'a polylist
type ('a, 'b) pair = Pair of 'a *'b
```

To simplify the compilation process we prohibit the `unit` type in datatype definitions. This does not cause any loss of generality since the excluded datatypes are isomorphic to types of the kind which we do allow. Values belonging to user-defined types are created by applying constructors and are deconstructed using the `match` statement:

```
let rec length l = match l with
       Nil -> 0
      | Cons (h,t) -> 1+length t

let test () = let l = Cons(2, Cons(7,Nil))
    in length l
```

The form of the match statement is much more restricted than in SML or O'Caml. There must be exactly one rule for each constructor in the associated datatype, and each rule binds the values contained in the constructor to variables (or discards them by using the pseudo-variable _). Complex patterns are not available, and must be simulated with further `match` and `if` statements.

As can be seen from the example above, constructor arguments are enclosed in parentheses and are separated by commas. In contrast, function definitions and applications which require multiple arguments are written in a "curried" style:

```
let add a b = a+b
let f x y z = add x (add y z)
```

Despite this notation, the present version of Camelot does *not* support higher-order functions; any application of a function must involve exactly the same number of arguments as are specified in the definition of the function.

3.2.2 Diamonds and Resource Control

Our current implementation of Camelot targets the Java Virtual Machine, and values from user-defined datatypes are represented by heap-allocated objects from a certain JVM class. Details of this representation will be given in Sec. 3.4.1.

Consider the following function which uses an accumulator to reverse a list of integers (as defined by the `intlist` type above).

```
let rec rev l acc = match l with
    Nil -> acc
   | Cons (h,t) -> rev t (Cons (h,acc))
let reverse l = rev l Nil
```

This function allocates an amount of memory equal to the amount occupied by the input list. If no further reference is made to the input list then the heap space which it occupies may eventually be reclaimed by the JVM garbage collector.

In order to allow more precise control of heap usage, Camelot includes constructs allowing re-use of heap cells. There is a special type known as the *diamond type* (denoted by <>) whose values represent blocks of heap-allocated memory, and Camelot allows explicit manipulation of diamond objects. This is achieved by equipping constructors and match rules with special annotations referring to diamond values. Here is the `reverse` function rewritten using diamonds so that it performs in-place reversal:

```
let rec rev l acc = match l with
    Nil -> acc
   | Cons (h,t)@d -> rev t (Cons (h,acc)@d)
let reverse l = rev l Nil
```

The annotation “@d” on the first occurrence of Cons tells the compiler that the diamond value d is to be bound to a reference to the space used by the list cell. The annotation on the second occurrence of Cons specifies that the list cell Cons(h,acc) should be constructed in the diamond object referred to by d, and no new space should be allocated on the heap.

One might not always wish to re-use a diamond value immediately. This can sometimes cause difficulty since such diamonds might then have to be returned as part of a function result so that they can be recycled by other parts of the program. For example, the alert reader may have noticed that the list reversal function above does not in fact reverse lists entirely in place. When the user calls reverse, the invocation of the Nil constructor in the call to rev will cause a new list cell to be allocated. Also, the Nil value at the end of the input list occupies a diamond, and this is simply discarded in the second line of the rev function (and will be subject to garbage collection if there are no other references to it). The overall effect is that we create a new diamond before calling the rev function and are left with an extra diamond after the call has completed. We could recover the extra diamond by making the reverse function return a pair consisting of the reversed list and the spare diamond, but this is rather clumsy and programs quickly become very complex when using this sort of technique. To avoid this kind of problem, unwanted diamonds can be stored on a *freelist* for later use. This is done by using the annotation “@_” as in the following example which returns the sum of the entries in an integer list, destroying the list in the process:

```
let rec sum l acc = match l with
    Nil@_ -> acc
  | Cons (h,t)@_ -> sum t (acc+h)
```

The question now is how the user retrieves a diamond from the freelist. In fact, this happens automatically during constructor invocation. If a program uses an undecorated constructor such as Nil or Cons(4,Nil) then if the freelist is empty the JVM new instruction is used to allocate memory for a new diamond object on the heap; otherwise, a diamond is removed from the head of the freelist and is used to construct the value. It may occasionally be useful to explicitly return a diamond to the freelist and an operator free: <> -> unit is provided for this purpose.

There is one final notational refinement. The in-place list reversal function above is still not entirely satisfactory since the Nil value carries no data but is nonetheless allocated on the heap. We can overcome this by redefining the intlist type as

```
type intlist = !Nil | Cons of int * intlist
```

The exclamation mark directs the compiler to represent the Nil constructor by the JVM null reference. With the new definition of intlist the original list-reversal function performs true in-place reversal: no heap space is consumed or destroyed when the reverse function is applied. The ! annotation can be used for a single zero-argument constructor in any datatype definition. In addition, if every constructor for a particular datatype is nullary then they may all be preceded by !, in which case they will be represented by integer values at runtime. We have

deliberately chosen to expose this choice to the programmer (rather than allowing the compiler to automatically choose the most efficient representation) in keeping with our policy of not allowing the compiler to perform optimisations which have unexpected results on resource consumption.

The features described above are very powerful and can lead to many kinds of program error. For example, if one applied the `reverse` function to a sublist of some larger list then the small list would be reversed properly, but the larger list could become partially reversed. Perhaps worse, a diamond object might be used in several different data structures of different types simultaneously. Thus a list cell might also be used as a tree node, and any modification of one structure might lead to modifications of the other. The simplest way of preventing this kind of problem is to require linear usage of heap-allocated objects, which means that variables bound to such objects may be used at most once after they are bound. Details of this approach can be found in Hofmann's paper [6]. Strict linearity would require one to write the list length function as something like

```
let rec length l = match l with
      Nil -> Pair (0, Nil)
    | Cons(h,t)@d ->
      let p = length t
      in match p with
           Pair(n, t1)@d1 -> Pair(n+1, Cons(h,t1)@d)@d1
```

It is necessary to return a new copy of the list since it is illegal to refer to `l` after calling `length l`.

Our compiler has a switch to enforce linearity, but the example demonstrates that the restrictive nature of linear typing can lead to unnecessary complications. Aspinall and Hofmann [1] give a type system which relaxes the linearity condition while still allowing safe in-place updates, and Michal Konečný generalises this still further in [9, 10]. As part of the MRG project, Konečný has implemented a typechecker for a variant of the type system of [9] adapted to Camelot.

A different approach to providing heap-usage guarantees is given by Hofmann and Jost in [7], where an algorithm is presented which can be used to statically infer heap-usage bounds for functional programs of a suitable form. In collaboration with the MRG project, Steffen Jost has implemented a variant of this inference algorithm for Camelot. The implementation is described in [8].

Both of these implementations are currently stand-alone programs, but we are in the process of integrating them with the Camelot compiler.

One of our goals in the design of Camelot was to define a language which could be used as a testbed for different heap-usage analysis methods. The inclusion of explicit diamonds fits the type systems of [1, 9, 10], and the inclusion of the freelist facilitates the Hofmann-Jost inference algorithm, which requires that all memory management takes place via a freelist. We believe that the fact that implementations of two radically different systems have been based on Camelot indicates that our goal was achieved successfully.

3.3 GRAIL

Instead of translating directly to JVM bytecode, the Camelot compiler targets the intermediate language Grail (Guaranteed resource allocation intermediate language). This is a small typed language which allows us to represent (a subset of) JVM bytecode in a functional form (see [13] or [23] for more information about the Java Virtual Machine and JVM bytecode). The design of Grail was inspired by the λJVM language of [11]. We will give a brief overview of Grail here. For further details see [14] or [3].

A Grail program defines a single Java class, potentially containing static fields, instance fields, static methods and instance methods. Field definitions are straightforward. The real interest of Grail lies in method definitions, which are represented in a functional form whose syntax is given below.

$$
\begin{array}{rcl}
\mathit{methoddef} & ::= & \texttt{method}\ \mathit{modifiers}\ \mathit{rty}\ \mathit{jname}\ \texttt{(}\langle \mathit{ty}_1\ \mathit{var}_1, \cdots, \mathit{ty}_n\ \mathit{var}_n\rangle\texttt{)}\ \texttt{=}\ \mathit{methodbody} \\
\mathit{methodbody} & ::= & \texttt{let}\ \langle \mathit{valdec}_1 \cdots \mathit{valdec}_m\rangle\ \langle \mathit{fundec}_1 \cdots \mathit{fundec}_n\rangle\ \texttt{in}\ \mathit{result}\ \texttt{end} \\
\mathit{valdec} & ::= & \texttt{val}\ \mathit{var}\ \texttt{=}\ \mathit{primop} \mid \texttt{val ()}\ \texttt{=}\ \mathit{primop} \\
\mathit{fundec} & ::= & \texttt{fun}\ \mathit{fname}\texttt{(}\langle \mathit{ty}_1\ \mathit{var}_1, \cdots, \mathit{ty}_n\ \mathit{var}_n\rangle\texttt{)}\ \texttt{=}\ \mathit{funbody} \\
\mathit{funbody} & ::= & \mathit{result} \mid \texttt{let}\ \langle \mathit{valdec}_1 \cdots \mathit{valdec}_n\rangle\ \texttt{in}\ \mathit{result}\ \texttt{end} \\
\mathit{result} & ::= & \mathit{primres} \mid \texttt{if}\ \mathit{value}\ \mathit{test}\ \mathit{value}\ \texttt{then}\ \mathit{primres}\ \texttt{else}\ \mathit{primres} \\
\mathit{primres} & ::= & \mathit{primop} \mid \texttt{()} \mid \mathit{fname}\texttt{(}\langle \mathit{var}_1, \cdots, \mathit{var}_n\rangle\texttt{)} \\
\mathit{primop} & ::= & \mathit{value} \mid \mathit{binop}\ \mathit{value}\ \mathit{value} \mid \texttt{new}\ \texttt{<}\mathit{condesc}\texttt{>}\ \texttt{(}\langle \mathit{value}_1, \cdots, \mathit{value}_n\rangle\texttt{)} \\
 & & \mid \texttt{invokevirtual}\ \mathit{var}\ \texttt{<}\mathit{methoddesc}\texttt{>}\ \texttt{(}\langle \mathit{value}_1, \cdots, \mathit{value}_n\rangle\texttt{)} \\
 & & \mid \texttt{invokestatic}\ \texttt{<}\mathit{methoddesc}\texttt{>}\ \texttt{(}\langle \mathit{value}_1, \cdots, \mathit{value}_n\rangle\texttt{)} \\
 & & \mid \texttt{invokespecial}\ \mathit{var}\ \texttt{<}\mathit{methoddesc}\texttt{>}\ \texttt{(}\langle \mathit{value}_1, \cdots, \mathit{value}_n\rangle\texttt{)} \\
 & & \mid \texttt{getfield}\ \mathit{var}\ \texttt{<}\mathit{fielddesc}\texttt{>} \mid \texttt{putfield}\ \mathit{var}\ \texttt{<}\mathit{fielddesc}\texttt{>}\ \mathit{value} \\
 & & \mid \texttt{getstatic}\ \texttt{<}\mathit{fielddesc}\texttt{>} \mid \texttt{putstatic}\ \texttt{<}\mathit{fielddesc}\texttt{>}\ \mathit{value} \\
 & & \mid \texttt{checkcast}\ \mathit{longjname}\ \mathit{var} \mid \texttt{instanceof}\ \mathit{longjname}\ \mathit{var} \\
 & & \mid \texttt{itof}\ \mathit{value} \mid \texttt{ftoi}\ \mathit{value} \mid \mathit{arrayop} \\
\mathit{arrayop} & ::= & \texttt{empty}\ \mathit{value}\ \mathit{ty} \mid \texttt{length}\ \mathit{var} \mid \texttt{get}\ \mathit{var}\ \mathit{value} \mid \texttt{set}\ \mathit{var}\ \mathit{value}\ \mathit{value} \\
\mathit{condesc} & ::= & \mathit{longjname}\ \texttt{(}\langle \mathit{ty}_1, \cdots, \mathit{ty}_n\rangle\texttt{)} \\
\mathit{methoddesc} & ::= & \mathit{rty}\ \mathit{longjname}\ \texttt{(}\langle \mathit{ty}_1, \cdots, \mathit{ty}_n\rangle\texttt{)} \\
\mathit{fielddesc} & ::= & \mathit{ty}\ \mathit{longjname} \\
\mathit{test} & ::= & \texttt{=} \mid \texttt{<>} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\mathit{binop} & ::= & \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{div} \mid \texttt{mod} \\
\mathit{value} & ::= & \mathit{var} \mid \mathit{intvalue} \mid \mathit{floatvalue} \mid \mathit{stringvalue} \mid \texttt{null}\,\texttt{[}\mathit{longjname}\texttt{]} \\
\mathit{ty} & ::= & \texttt{int} \mid \texttt{float} \mid \texttt{string} \mid \mathit{longjname} \mid \mathit{ty}\texttt{[]} \\
\mathit{rty} & ::= & \mathit{ty} \mid \texttt{void} \\
\mathit{modifiers} & ::= & \langle \texttt{public} \mid \texttt{protected} \mid \texttt{private}\rangle\ \langle \texttt{static}\rangle\ \langle \texttt{final}\rangle
\end{array}
$$

The terms *longjname* and *jname* refer to Java-style class, field and method names; items of type *longjname* may contain dots, whereas those of type *jname* may

not. In addition, Java method names for initialisers may end with `.<init>` or `.<clinit>`. The terms *var* and *fname* denote local variable names and function names respectively. Expressions of the form `val () = ...` are used to invoke operations such as `putfield` which do not return a result, and also to call `void` methods.

As a simple example of Grail, the following code defines a class containing a method for calculating the factorial of an integer.

```
class Fac {
   method public static int fac (int n) =
   let
      val b = 1
      fun f(int n, int b) = if n < 1 then b else f_else(n,b)
      fun f_else(int n, int b) =
      let val b = mul b n
          val n = sub n 1
      in f(n,b) end
   in f(n,b) end
}
```

3.3.1 The Grail Type System

Grail implements a type system similar to a subset of the JVM type system. The `int` and `float` types are the same as corresponding JVM types. There is also a collection of *reference types* which represent Java class instances and arrays. These can be used to access any Java class or method from Grail. The concrete syntax also includes a `string` type which is the same as `java.lang.String`. One major difference between the Grail and JVM type systems is that there is no subtyping in Grail. The JVM allows an object x from a class C to be used in any context where an object from a superclass S of C is expected, but Grail requires object types to match exactly. The object x must be explicitly upcast to S using the `checkcast` operation before the assignment takes place. This causes unnecessary casting operations to occur in the corresponding bytecode, but enables considerable simplifications in typechecking for Grail; furthermore, the Camelot compiler does not make any use of the Java inheritance features at present, so this point does not cause any problems.

3.3.2 Compilation of Grail

We will describe some features of the Grail compilation process. Full details can be found in [14].

In Grail, named variables are in one-to-one correspondence with JVM local variables. The JVM operand stack is used in a very restricted way in that intermediate results may not be left on the stack for later use: they must immediately be stored in a variable, leaving the stack empty. Thus to add three integers one must add the first two and store the result in some intermediate variable x, say, and then add the final variable to x.

The primitive operations (the class *primop* above) correspond directly to atomic JVM operations and can be translated more or less verbatim (except for the Grail `new` operation, which combines object creation and initialisation).

Each Grail method is compiled into a JVM method. The JVM is an imperative machine with branches and `goto` statements, but these instructions are not visible in Grail. Instead, flow control within a Grail method is handled by calls to local functions defined in the method. These function calls are very restricted: they may only occur in tail position, and we require that whenever a function is called the names of its actual parameters must exactly match those used in its declaration. This convention allows a very simple translation to JVM bytecode. Function bodies are translated into basic blocks of bytecode, and every function call may assume that its arguments are already stored in the correct registers, so that the call can be translated into a direct jump.

The structure of Grail (in particular the calling convention) means that there is a very close correspondence between functional Grail and the imperative bytecode obtained by compiling it. In fact, the resulting bytecode is so idiomatic that it is easy to translate it back to the original Grail source, which is a useful feature from the PCC viewpoint. In addition, the transparency of the correspondence is important from the point of view of resource accounting. For example, the calling convention means that no extra code which might affect execution time or stack size has to be introduced to place arguments in the correct registers.

The restricted form of Grail bytecode also has interesting implications for the JVM verification process. One example of this is that the structure of the language in fact guarantees that valid Grail will compile to verifiable bytecode (we do not have a formal proof of this, but we are confident that it is true); this means that we have a *syntactic* guarantee of verifiability, whereas the verifiability of arbitrary bytecode can only be established *algorithmically*, by actually running the verification algorithm.

It also turns out that bytecode obtained from Grail is much easier to verify than arbitrary bytecode. For example, one of the conditions that bytecode must satisfy during verification is that at any particular point in a program the number and types of the elements on the operand stack are independent of the path taken to reach that point. To establish this requires an iterative dataflow analysis to calculate fixpoints for stack types (see [13, Sec. 4.9.2]), which can consume a lot of time and space (see [12, Sec. 2.3] for some concrete figures). In [12] Leroy examines JVM bytecode verification in detail and shows that if some simple restrictions are imposed on the form of the bytecode (notably that the stack be empty at each jump destination) then checking this property is considerably simpler. In fact, Leroy shows that the entire verification process can be carried out in constant space (in practice, less than 100 bytes). The improvement is such that bytecode verification can be performed even with the extremely limited resources of a smartcard. This has hitherto been infeasible, and the standard approach has been for a trusted agent to perform off-card verification of bytecode prior to downloading. It is easily seen that Grail satisfies Leroy's conditions, which is encouraging since we hope to use it with devices with limited resources.

Some other properties of Grail are studied in [3]: among other things it is shown that the structure of Grail has connections with the well-known static single-assignment form.

We have implemented programs called `gdf` and `gf` which perform the translation from Grail to JVM and back. These can be downloaded from [15].

3.4 COMPILING CAMELOT TO GRAIL

We have implemented a Camelot compiler (available from [15]) which operates by translating Camelot into Grail and then into JVM bytecode. The compiler is a whole-program compiler whose back end is essentially the `gdf` program mentioned above. This section will describe the translation from Camelot to Grail.

3.4.1 Representing Data

Our compilation strategy is *type-preserving* in that well-typed Camelot programs are translated into well-typed Grail programs. This increases the robustness of the compiler since implementation errors often lead to type errors in the Grail code which are then detected by the Grail typechecker in the back end of the compiler.

The basic types `bool`, `int`, `float` and `string` are represented by the obvious Grail types. The `unit` type causes difficulties since there is no corresponding type in Grail. It is in fact possible to "compile away" occurrences of the unit type: this is described in an extended version of this paper available from [15].

Objects belonging to user-defined datatypes are represented by members of a single JVM class which we will refer to as the *diamond class*. Objects of the diamond class contain enough fields to represent any member of *any* datatype defined in the program. Each instance X of the diamond class contains an integer tag field which identifies the constructor with which X is associated. The diamond class also contains a static field pointing to the freelist. The freelist is managed via the static methods `alloc` (which returns the diamond at the head of the freelist, or creates a new diamond by calling `new` if the freelist is empty), and `free` which places a diamond object on the freelist. The diamond class also has overloaded static methods called `make` and `fill`, one instance of each for every sequence of types appearing in a constructor. The `make` methods are used to implement ordinary constructor application; each takes an integer tag value and a sequence of argument values and calls `alloc` to obtain an instance of the diamond class, and then calls a corresponding `fill` method to fill in the appropriate fields with the tag and the arguments. The `fill` methods are also used when the programmer reuses an existing diamond to construct a datatype value.

It can be argued that this representation is inefficient in that datatype values are often represented by JVM objects which are larger than they need to be. This is true, but is difficult to avoid owing to the type-safe nature of JVM memory management which prevents one from re-using the heap space occupied by a value of one type to store a value of a different type. We wish to be able to reuse heap space, but this can be impossible if objects can contain only one type of data.

With the current scheme one can easily write a heapsort program which operates entirely in-place. List cells are large enough to be reused as heap nodes and this allows a heap to be built using cells obtained by destroying the input list. Once the heap has been built it can in turn be destroyed and the space reused to build the output list. In this case, the amount of memory occupied by a list cell is larger than it needs to be, but the overall amount of store required is less than would be the case if separate classes were used to contain list cells and heap nodes.

In the current context it can be claimed that it is better to have an inefficient representation about which we can give concrete guarantees than an efficient one which about we can say nothing. Most of the programs which we have written so far use a limited number of datatypes so that the overhead introduced by the monolithic representation for diamonds is not too severe. However, it is likely that for very large programs this overhead would become unacceptably large. One possibility which we have not yet explored is that it might be possible to achieve more efficient heap usage by using dataflow techniques to follow the flow of diamonds through the program and detect datatypes which are never used in an overlapping way. One could then equip a program with several smaller diamond classes which would represent such non-overlapping types.

These problems could be avoided by compiling to some platform other than the JVM (for example to C or to a specialised virtual machine) where compaction of heap regions would be possible. The Hofmann-Jost algorithm is still applicable in this situation, so it would still be feasible to produce resource guarantees. However, it was a fundamental decision of the MRG project to use the JVM, based on the facts that the JVM is widely deployed and very well-known and that resource usage is a genuine concern in many contexts where the JVM is used. Our present approach allows us to produce concrete guarantees at the cost of some overhead; we hope that at a later stage a more sophisticated approach (such as the one suggested above) might allow us to reduce the overheads while still obtaining guaranteed resource bounds.

3.4.2 Compilation of Programs

We compile a Camelot program to a single class with one static method for each function in the program. This technique is somewhat problematic since recursive function calls translate to recursive calls on JVM methods, which are expensive and can potentially lead to overflow of the JVM stack.

Functions which call themselves in a tail-recursive manner can safely be compiled into recursive Grail function calls, and a compiler option is available which enables this feature (see [24], which also includes a proof that the optimisation has no effect on heap usage). However, mutually tail-recursive functions are difficult to program within a single stack frame because JVM methods can only have one entry point and there is a limit on the size of method bodies.

Various techniques are known which can overcome this problem (for example, the *trampoline* [22, §6.2], Baker's "Cheney on the MTA" technique [2]). Unfortunately, all of these strategies tend to require extra heap usage and thus compromise the transparency of the compilation process. Because of this, at present we sim-

ply compile each function as a separate method and implement (non-recursive) tail calls as standard method calls, which carries a risk of stack overflow in programs which make a lot of use of mutual recursion. We will return to this problem in our closing remarks.

3.4.3 Initial Transformations

Compilation begins with a phase in which several transformations are applied to the abstract syntax tree.

Monomorphisation

Firstly, all polymorphism is removed from the program. For polymorphic types $(\alpha_n, \ldots, \alpha_1)\ t$ such as α `list` we examine the entire program to determine all instantiations of the type variables and compile a separate datatype for each distinct instantiation. Similarly, whenever a polymorphic function is defined the program is examined to find all uses of the function and a monomorphic function of the appropriate type is generated for each distinct instantiation of types.

Normalisation

After monomorphisation there is a phase referred to as *normalisation* which transforms the Camelot program into a form (*Normalised Camelot*) which closely resembles Grail.

First, the compiler ensures that all variables have unique names. Any duplication is resolved by generating new names. This allows us to map Camelot variable names directly onto Grail variable names (which in turn map onto JVM local variable locations) with no danger of clashes arising.

We next have to simplify boolean expressions. Grail has no direct equivalent for expressions such as $m < n$ outside `if`-expressions and we deal with this by replacing such expressions with ones of the form `if` $m < n$ `then true else false`.

Next, we give names to intermediate results in many contexts by replacing complex expressions with variables. For example, the expression $f(a+b+c)$ would be replaced by an expression of the form `let` $t_1 = a + b$ `in let` $t_2 = t_1 + c$ `in` $f\ t_2$. The introduction of names for intermediate results can produce a large number of Grail (and hence JVM) variables. After the source code has been compiled to Grail the number of local variables is minimised by applying a standard register allocation algorithm (see [24]).

A final transformation ensures that `let`-expressions are in a "straight-line" form. After all of these transformations have been performed expressions have been reduced to the following form:

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \mathit{expr}' \mid \texttt{let}\ \mathit{pat} = \mathit{expr}'\ \texttt{in}\ \mathit{expr} \\
\mathit{expr}' & ::= & \mathit{primexp} \mid \texttt{if}\ \mathit{atom}\ \mathit{cmp}\ \mathit{atom}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \\
 & & \mid \texttt{if}\ \mathit{atom}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \mid \texttt{match}\ \mathit{var}\ \texttt{with}\ \mathit{match}\ \texttt{end}
\end{array}
$$

$$
\begin{array}{rcl}
\textit{primexp} & ::= & \textit{atom} \mid \textit{uop atom} \mid \textit{atom arithop atom} \mid \texttt{free}\ \textit{var} \\
& & \mid \textit{fname}\ \textit{atom}_1 \ldots \textit{atom}_n \mid \textit{cname}\ \texttt{(}\textit{atom}_1, \ldots, \textit{atom}_n\texttt{)}\ \langle \texttt{@}\textit{var} \rangle \\
\textit{atom} & ::= & \textit{const} \mid \textit{var}
\end{array}
$$

(undefined syntactic classes remain the same as those in the full syntax of Camelot given earlier). The structure of normalised Camelot (which is in fact in a type of A-normal form [5]) is sufficiently close to that of Grail to make it fairly easy to translate from the former to the latter. Another benefit of normalisation is that it is easier to write and implement type systems for normalised Camelot. The fact that the components of many expressions are atoms rather than complex subexpressions means that typing rules can have very simple premisses.

3.4.4 Compilation of Expressions

The Camelot expressions labelled by the term *primexp* in the normalised syntax above will be referred to as *primitive expressions*. They are significant because they correspond directly to primitive operations in Grail and thus admit an easy translation. This is the key to compilation of normalised Camelot into Grail. A normalised Camelot expression consists of a nested sequence of `let` expressions. The translation procedure essentially translates an expression (in particular, a function body) into a collection of mutually recursive Grail local functions by descending down the chain of `let`-expressions, emitting a Grail *valdec* for each term of the form `let` $p = e$ with e primitive. This process terminates when a non-primitive expression e is encountered; at this point e must be a branch of some kind, and the compiler recursively generates a new local function for each of the subexpressions occurring in the branch, terminating the original function with a Grail `if`-result (or, in the case of a `match` statement, a block of code implementing a sequence of such results). This a highly simplified description of the translation to Grail; space constraints preclude a full description, but the extended version of this paper (see [15]) contains an appendix giving a full and precise specification of the translation.

3.5 PERFORMANCE

We have described a procedure for compiling Camelot into Grail, and thence to JVM. This is a long process involving several different stages, and one might suspect that it would introduce inefficiencies into the final bytecode programs. In this section we will present figures comparing the run-time of various Camelot programs with versions of the same programs written in Java and in Scheme, which we hope will demonstrate that performance is not compromised unduly.

Java programs were compiled using the standard Sun Java compiler. To compile Scheme programs for the JVM the Bigloo Scheme compiler [20, 19] was used.

Timings were obtained using the JFluid JVM profiling tool [4]; this uses a special version of the Sun JVM (version 1.4.2) which has been modified to allow dynamic instrumentation of class files. The figures which are obtained appear to be fairly accurate since one can focus on particular areas of the program without incurring an overhead by profiling irrelevant code. By default the JVM performs adaptive compilation to native code for frequently-executed code sequences. This feature is not available in JFluid, so all execution was performed by interpretation. However, we felt that this would still give a realistic (worst-case) estimate of program times. Also, JVMs for limited-memory devices generally provide no alternatives to interpretation. The timings were carried out on a 366MHz Pentium 2 processor under Linux. All timings are in milliseconds and represent an average taken over five runs.

Firstly we consider several list-reversal programs. Each program generates a list consisting of the integers between 1 and 1,000,000 and then proceeds as follows:

- A reverses the list in place.
- B reverses the list in place, but replaces each element x by $x+x$.
- C returns a reversed copy of the list, leaving the original intact.
- D returns a reversed copy with each element doubled as in B.

We timed the execution of the entire program (including construction of the input list) and also of the reversal function in isolation. The results follow below.

	A		B	
	main	reverse	main	reverse
Java	6289ms	507ms	6653ms	850ms
Camelot	11263ms	1684ms	11684ms	1785ms
Scheme	28884ms	3645ms	58595ms	30734ms

	C		D	
	main	reverse	main	reverse
Java	10824ms	5009ms	10670ms	5215ms
Camelot	20285ms	10439ms	20580ms	10676ms
Scheme	31686ms	6829ms	54178ms	28822ms

We note that the Camelot versions are slower than the Java versions but are generally faster than the Scheme versions. There are several reasons why Camelot is slower than Java.

(1) The requirement that all intermediate results in Grail are explicitly named means that the bytecode often contains pairs of instructions where a value is stored in a local variable and then immediately recalled for further use (and the stored copy is never used again). This certainly has the effect of slowing down the execution of the bytecode, but the decision to use this form of code was made deliberately in the hope that the regularity of the bytecode would simplify formal analyses.

(2) In Camelot it is not possible to modify individual fields within an object: when a value is constructed in a recycled Camelot diamond, the fields in the corresponding object are filled in by a method call (to the `fill` method mentioned in 3.4.1). All fields must be explicitly rewritten, even if some have not changed (see the reversal example in 3.2.2, which is essentially the same as the one used in program A). In contrast, in Java one can perform list-reversal simply by changing pointers in list cells and leaving the other values stored in the cells intact. This accounts for the fact that simple in-place reversal in Java is three times as fast as in Camelot, but when the entries in the list are modified, as in program B, the Java version is only twice as fast as the Camelot version. The fact that a method call is used, rather than a sequence of `putfield` operations, also adds some extra overhead. Again, this was a conscious design decision: a constructor application in Camelot corresponds directly to a single method application, and it was felt that this correspondence would simplify analysis.

We performed the Scheme comparisons as we thought it would be interesting to compare Camelot's performance with that of another functional language running on the JVM. It was somewhat surprising to discover that while Scheme took six times as long as Java to perform simple in-place list reversal, it took more than 36 times as long to perform reversal with doubling. This appears to be due to the fact that Scheme's numeric `+` operator is overloaded. Inspection of the bytecode produced by the compiler reveals that Bigloo handles overloading by representing numeric values in a boxed form as Java objects. When elements in the list are doubled this requires the `+` operator to examine the boxed values to determine their numeric type, then to call an appropriate specialised addition operator, and finally to re-box the result prior to insertion in the modified list. Since this happens for each of the million elements in the list it is not surprising that there is a considerable slowdown. By using the Scheme `+fx` operator in place of `+` it is possible to use Scheme `fixnum` values, which Bigloo encodes as JVM `int` values. When program B is modified in this way the execution time for the reversal function reduces to about 14000ms. This figure is still about 16 times as long as the Java version: we suspect that this is largely due to the fact that dynamic typechecking is still required before the addition operator is actually called.

The following table gives timings for some other programs:

	Fibonacci	Quicksort	Insertion Sort
Java	221229ms	21009ms	23963ms
Camelot	239039ms	34166ms	42415ms
Scheme	709598ms	42368ms	73412ms

The first column gives times for calculation of the 40th Fibonacci number by a direct implementation of the recursive definition. Execution of the program consists mostly of recursive method invocations, so the performance of Java and Camelot is very similar. Again Scheme performs badly owing to dynamic typechecking. The figures given represent a calculation using `finxum` values; when these were replaced by the default boxed integer values, the execution time rose to 6577734ms, or about 1 hour and 49 minutes.

The second column of the table gives times for execution of an in-place quick-sort algorithm on a list of 25586 words (the text of [21]), and the third column gives times for an in-place insertion sort of a list consisting of the first 5000 words of the same list. Again Java performs best, with Camelot second and Scheme third, but in these examples the differences are less marked than in some of the previous examples.

Overall the figures show that Camelot programs compare favourably with Java programs. Furthermore, it is fairly clear which features of Camelot are responsible for its poorer performance. As suggested above, the somewhat rigid structure of the bytecode obtained from Camelot programs is due to deliberate design decisions which were made in order to allow a precisely-defined and transparent compilation procedure which would facilitate program analysis. It is possible that some of these restrictions could eventually be relaxed (thereby improving performance) without compromising the validity of our analyses.

We have only considered execution time here. Of course, our main interest is in memory usage. JFluid also allows one to collect memory profiling information, and this indicates that the heap usage of the Java and Camelot programs was exactly as expected. Unfortunately we were unable to obtain any heap profiling for the Scheme programs since they appeared to terminate in a nonstandard way which the JFluid system was unable to deal with properly.

3.6 FINAL REMARKS

We have described a technique for compiling Camelot into JVM bytecode via the functional intermediate language Grail; we believe that this technique satisfies the strict requirements of the PCC framework. We have also provided some performance figures which indicate that the rigid specification of the compilation procedure does not degrade execution speed unduly.

There are various ways in which Camelot could be extended. The lack of higher-order functions is inconvenient, but the resource-aware type systems which we use are presently unable to deal with higher-order functions, partly because of the fact that these are normally implemented using heap-allocated closures whose size may be difficult to predict. A possible strategy for dealing with this which we are currently investigating is Reynolds' technique of *defunctionalization* [18] which transforms higher-order programs into first-order ones, essentially by performing a transformation of the source code which replaces closures with members of datatypes. This has the advantage that extra space required by closures is exposed at the source level, where it is amenable to analysis by the heap-usage inference techniques mentioned earlier.

A similar strategy can be used to eliminate mutual tail-recursion. Given a set of mutually recursive functions $\mathcal{F}$ whose results are of type `t`, we define a datatype `s` which has for each of the functions in $\mathcal{F}$ a constructor with arguments corresponding to the function's arguments. The collection of functions $\mathcal{F}$ is then replaced by a single function `f: s -> t` whose body is a `match` statement which carries out the computations required by the individual functions in $\mathcal{F}$. In this

way the mutually recursive functions can be replaced by a single tail-recursive function, and we already have an optimisation which eliminates recursion for such functions. This technique is somewhat clumsy and care is required in recycling the diamonds which are required to contain members of the datatypes required by `s`. Another potential problem is that several small functions are effectively combined into one large one, and there is thus a danger that that 64k limit for JVM methods might be exceeded. Nevertheless, this technique does overcome the problems related to mutual recursion without affecting the transparency of the compilation process unduly, and it might be possible for the compiler to perform the appropriate transformations automatically. We intend to investigate this in more detail.

As an extension in a different direction, the second author has recently extended the language (and the compiler) to include object-oriented features and allow the use of pre-existing Java libraries: details can be found in [25].

As mentioned earlier, complex resource-aware type-systems and inference methods have been implemented for Camelot and will soon be integrated with the present compiler. Eventually, the MRG project aims to have a certifying compiler which will take a Camelot program and automatically provide a proof that it abides by a given resource policy.

Acknowledgments

The authors would like to thank Hans-Wolfgang Loidl and Ian Stark for their comments.

This research was supported by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing.

REFERENCES

[1] David Aspinall and Martin Hofmann. Another type system for in-place update. In *Proc. 11th European Symposium on Programming, Grenoble*, volume 2305 of *Lecture Notes in Computer Science*. Springer, 2002.

[2] Henry G. Baker. CONS should not CONS its arguments, part II: Cheney on the M.T.A. *ACM SIGPLAN Notices*, 30(9):17–20, September 1995.

[3] Lennart Beringer, Kenneth MacKenzie, and Ian Stark. Grail: a functional form for imperative mobile code. In Vladimiro Sassone, editor, *Electronic Notes in Theoretical Computer Science*, volume 85. Elsevier, 2003.

[4] M. Dmitriev. Welcome to JFluid, October 2003. Documentation and download available at http://research.sun.com/projects/jfluid.

[5] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings ACM SIGPLAN 1993 Conf. on Programming Language Design and Implementation, PLDI'93, Albuquerque, NM, USA, 23–25 June 1993*, volume 28(6), pages 237–247. ACM Press, New York, 1993.

[6] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[7] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. 30th ACM Symp. on Principles of Programming Languages, New Orleans*, 2003.

[8] S. Jost. `lfd_infer`: an implementation of a static inference on heap-space usage. In *Proceedings of SPACE'04, Venice*, 2004. To appear.

[9] Michal Konečný. Functional in-place update with layered datatype sharing. In *TLCA 2003, Valencia, Spain, Proceedings*, pages 195–210. Springer-Verlag, 2003. Lecture Notes in Computer Science 2701.

[10] Michal Konečný. Typing with conditions and guarantees for functional in-place update. In *TYPES 2002 Workshop, Nijmegen, Proceedings*, pages 182–199. Springer-Verlag, 2003. Lecture Notes in Computer Science 2646.

[11] Christopher League, Valery Trifonov, and Zhong Shao. Functional Java bytecode. In *Proc. 5th World Conf. on Systemics, Cybernetics, and Informatics*, July 2001. Workshop on Intermediate Representation Engineering for the Java Virtual Machine.

[12] Xavier Leroy. Bytecode verification for Java smart cards. *Software Practice & Experience*, 32:319–340, 2002.

[13] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999. Available at http://java.sun.com/docs/books/vmspec/.

[14] K. MacKenzie. Grail: a functional intermediate language for resource-bounded computation. LFCS, University of Edinburgh, 2002. Available at http://www.lfcs.inf.ed.ac.uk/mrg/publications/.

[15] The Mobile Resource Guarantees project. http://www.lfcs.inf.ed.ac.uk/mrg.

[16] George C. Necula. Proof-carrying code. In *Conference Record of POPL '97: 24th ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[17] O'Caml. Welcome to the O'Caml language, October 2003. See www.ocaml.org.

[18] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, December 1998.

[19] M. Serrano. See http://www-sop.inria.fr/mimosa/fp/Bigloo.

[20] Manuel Serrano and Pierre Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *Static Analysis Symposium*, pages 366–381, 1995.

[21] Robert Louis Stevenson. *Strange Case of Dr. Jekyll and Mr. Hyde*. Longmans, Green, London, 1886. Available online at http://www.gutenberg.org.

[22] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: compiling standard ML to C. *ACM Letters on Programming Languages and Systems*, 1(2):161–177, June 1992.

[23] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, second edition, 1999. Also at http://www.artima.com/insidejvm/blurb.html.

[24] N. Wolverson. Optimisation and resource bounds in Camelot compilation. Final-year project report, University of Edinburgh, 2003. Available at http://www.lfcs.inf.ed.ac.uk/mrg/publications/wolverson.ps.

[25] N. Wolverson and K. MacKenzie. O'Camelot: adding objects to a resource-aware functional language. In *Trends in Functional Programming Volume 4: Proceedings of TFP2003*, pages 47–62. Intellect, 2004.

Chapter 4

O'Camelot: Adding Objects to a Resource-Aware Functional Language

Nicholas Wolverson and Kenneth MacKenzie[1]

Abstract We outline an object-oriented extension to Camelot, a functional language in the ML family designed for resource aware computation. Camelot is compiled for the Java Virtual Machine, and our extension allows Camelot programs to interact easily with the Java object system, harnessing the power of Java libraries and allowing Java programs to incorporate resource-bounded Camelot code.[2]

4.1 INTRODUCTION

The Mobile Resource Guarantees (MRG) project aims to equip mobile bytecode programs with guarantees that their usage of certain computational resources (such as time, heap space or stack space) does not exceed some agreed limit, using a Proof Carrying Code framework. Programs written in the functional language Camelot will be compiled into bytecode for the Java Virtual Machine. The resulting class files will be packaged with a proof of the desired property and transmitted across the network to a code consumer—perhaps a mobile phone, or PDA. The recipient can then use the proof to verify the given property of the program before execution. There is thus an unforgeable guarantee that the program will not exceed the stated bounds.

The core Camelot language, as described in [8], enables the programmer to

[1]Laboratory for Foundations of Computer Science, The University of Edinburgh.
Email: `N.Wolverson@ed.ac.uk`, `kwxm@inf.ed.ac.uk`

[2]This research was supported by the MRG project (IST-2001-33149) which is funded by the EC under the FET proactive initiative on Global Computing.

write a program with a predictable resource usage; future work will provide each program with a proof that it does not exceed a stated resource bound. A compiler exists for this language, compiling polymorphic resource-aware Camelot programs to the JVM. However, only primitive interaction with the outside world is possible, through command line arguments, file input and printed output. To be able to write a full interface for a game or utility to be run on a mobile device, Camelot programs must be able to interface with external Java libraries. Similarly, the programmer may wish to utilise device-specific libraries or Java's extensive class library.

Here we describe an Object-Oriented extension to Camelot primarily intended to allow Camelot programs to access Java libraries. It would also be possible to write resource-certified libraries in Camelot for consumption by standard Java programs or indeed use the object system for OO programming for its own sake, but giving Camelot programs access to the outside world is the main objective.

4.2 CAMELOT

Camelot is an ML-like language with additional features to enable close control of heap usage. Certain restrictions are made in order to enable a compilation process which is transparent in terms of resource usage and to allow analysis of resource usage by various novel type systems.

The concrete syntax of Camelot is very close to O'Caml, as described in [1]. The following program defines a polymorphic list datatype and functions `sort` and `insert` performing an insertion sort on such lists.

```
type 'a lst = !Nil | Cons of 'a * 'a lst
let rec insert n l d =
  match l with Nil -> Cons(n, Nil)@d
             | Cons(h,t)@d' ->
               if n <= h then Cons(n, Cons(h,t)@d')@d
               else Cons(h, insert n t d)@d'
let rec sort l =
  match l with Nil -> Nil
             | Cons(h,t)@d -> insert h (sort t) d
```

Ignoring annotations such as `@d` and occurrences of the associated variable `d`, and the `!` in front of `Nil`, this program is valid O'Caml and indeed defines an insertion sort. Here we are more concerned about space rather than time issues; notice that the datatype constructor `Cons` is applied $O(n^2)$ times on average, but this much storage is not necessary. While a sensible garbage collector means we will not really lose the use of this space, this is not guaranteed, and we cannot predict when the space will be reclaimed. This is unacceptable when considering proof carrying code, and indeed on some mobile devices we will not have the luxury of a garbage collector at all.

In order to allow better control of heap usage, Camelot adds features allowing control of heap allocated storage. Camelot includes a *diamond type* (denoted by

<>) representing regions of heap-allocated memory and allows explicit manipulation of diamond objects. The representation of Camelot datatypes is critical here—values from user-defined datatypes are represented by heap-allocated objects from a certain Java class and a diamond value corresponds directly to an object of this class.

The diamond annotations in the above program result in an in-place insertion sort algorithm. During the execution of `sort` on a list, no new block of heap storage is allocated, but instead the existing storage is reused for the new list. The annotation `@d` on the occurrence of `Cons` in `sort` indicates that the space used in that list cell should be made available for re-use via the diamond value `d`. This diamond value is passed to a call of `insert`, where it is used in the expression `Cons(n, Nil)@d` to specify that the cons cell should be constructed in the heap space referred to by `d`. Lastly the use of `!` in the definition of the `Nil` constructor indicates that `Nil` does not take up a diamond (`Nil` is represented by the null pointer).

With explicit management of heap-space comes the possibility of program errors. The above sort function destroys the original list, so any subsequent attempt to reuse that list may result in an error, and if the list is a sublist of a larger list, the sublist will be correctly sorted but the larger list will become damaged. Various type systems can be used to ensure that diamond annotations are safe. Most simply, we can require all uses of heap-allocated storage to be linearly typed as described in [5]; the above program is typable under this system. We can also take a less restrictive approach as described in [7]. It is also possible to infer some diamond annotations, as shown in [6], and indeed this process can also give an upper bound on a program's heap usage.

As well as adding resource-related extensions, we make some restrictions, the first of which is to the form of patterns in the `match` statement. Nested patterns are not permitted, and instead each constructor of a datatype must be matched by exactly one pattern. Patterns are also not permitted in the arguments of function definitions. These features must be simulated by nested `match` statements.

The second restriction is on function application. While function application is written using a curried syntax as above, higher order functions are not permitted in the current version of Camelot. Functions must always be fully applied, and there is no lambda term. This is because closures would seem to introduce an additional non-transparent memory usage, although hopefully this can be overcome at a later date, and higher order functions added to the language.

4.3 EXTENSIONS

In designing an object system for Camelot, many choices are made for us, or are at least tightly constrained. We wish to create a system allowing inter-operation with Java, and we wish to compile an object system to JVML. So we are almost forced into drawing the object system of the JVM up to the Camelot level and cannot seriously consider a fundamentally different system.

On the other hand, the type system is strongly influenced by the existing

Camelot type system. There is more scope for choice, but implementation can become complex, and an overly complex type system is undesirable from a programmer's point of view. We also do not want to interfere with type systems for resources as mentioned above.

We shall first attempt to make the essential features of Java objects visible in Camelot in a simple form, with the view that a simple abbreviation or module system can be added at a later date to make things more palatable if desired.

Basic Features

We shall view objects as records of possibly mutable fields together with related methods, although Camelot has no existing record system. We define the usual operations on these objects, namely object creation, method invocation, field access and update, and casting and matching. As one might expect, we choose a class-based system closely modelling the Java object system. Consequently we must acknowledge Java's uses of classes for encapsulation and associate static methods and fields with classes also.

We now consider these features. The examples below illustrate the new classes of expressions we add to Camelot.

Static method calls There is no conceptual difference between static methods and functions, ignoring the use of classes for encapsulation, so we can treat static method calls just like function calls.

```
java.lang.Math.max a b
```

Static field access Some libraries require the use of static fields. We should only need to provide access to constant static fields, so they correspond to simple values.

```
java.math.BigInteger.ONE
```

Object creation We clearly need a way to create objects, and there is no need to deviate from the `new` operator. By analogy with standard Camelot function application syntax (i.e. curried form) we have:

```
new java.math.BigInteger "101010" 2
```

Instance field access To retrieve the value of an instance variable, we write

```
object#field
```

whereas to update that value we use the syntax

```
object#field <- value
```

assuming that `field` is declared to be a *mutable* field.

It could be argued that allowing unfettered external access to an object's variables is against the spirit of OO and, more to the point, inappropriate for our small language extension, but we wish to allow easy interoperability with any external Java code.

Method invocation Drawing inspiration from the O'Caml syntax, and again using a curried form, we have instance method invocation:

```
myMap#put key value
```

Null values In Java, any method with object return type may return the `null` object. For this reason we add a construct

`isnull` *e*

which tests if the expression *e* is a `null` value.

Casts and typecase It may occasionally be necessary to cast objects up to superclasses, for example to force the intended choice between overloaded methods. We will also want to recover subclasses, such as when removing an object from a collection. Here we propose a simple notation for up-casting:

```
obj :> Class
```

This notation is that of O'Caml, also borrowed by MLj (described in [2]). To handle down-casting we shall extend patterns in the manner of `typecase` (again like MLj) as follows:

```
match obj with o :> C1 -> o.a()
             | o :> C2 -> o.b()
             | _ -> obj.c()
```

Here `o` is bound in the appropriate subexpressions to the object `obj` viewed as an object of type `C1` or `C2` respectively. As in datatype matches, we require that every possible case is covered; here this means that the default case is mandatory. We also require that each class is a subclass of the type of `obj`, and suggest that a compiler warning should be given for any redundant matches.

Unlike MLj we choose not to allow downcasting outside of the new form of `match` statement, partly because at present Camelot has no exception support to handle invalid down-casts.

As usual, the arguments of a (static or instance) method invocation may be subclasses of the method's argument types, or classes implementing the specified interfaces.

The following example demonstrates some of the above features and illustrates the ease of interoperability. We will discuss the need for type constraints as on the parameter *l* later.

```
let convert (l: string list) =
  match l with [] -> new java.util.LinkedList ()
             | h::t ->
               let ll = convert t
               in let _ = ll#addFirst h
               in ll
```

Defining classes

Once we have the ability to write and compile programs using objects, we may as well start writing classes in Camelot. We must be able to create classes to implement callbacks, such as in the Swing GUI system which requires us to write stateful adaptor classes. Otherwise, as mentioned previously, we may wish to write Camelot code to be called from Java, for example to create a resource-certified library for use in a Java program, and defining a class is a natural way to do this. Implementation of these classes will obviously be tied to the JVM, but the form these take in Camelot has more scope for variation.

We allow the programmer to define a class which may explicitly subclass another class, and implement a number of interfaces. We also allow the programmer to define (possibly mutable) fields and methods, as well as static methods and fields for the purpose of creating a specific class for interfacing with Java. We naturally allow reference to `this`.

The form of a class declaration is given below. Items within angular brackets $\langle \ldots \rangle$ are optional.

$$
\begin{array}{rcl}
\mathit{classdecl} & ::= & \texttt{class}\ \mathit{cname} = \langle \mathit{scname}\ \texttt{with} \rangle\ \mathit{body}\ \texttt{end} \\
\mathit{body} & ::= & \langle \mathit{interfaces} \rangle\ \langle \mathit{fields} \rangle\ \langle \mathit{methods} \rangle \\
\mathit{interfaces} & ::= & \texttt{implement}\ \mathit{iname}\ \langle \mathit{interfaces} \rangle \\
\mathit{fields} & ::= & \mathit{field}\ \langle \mathit{fields} \rangle \\
\mathit{methods} & ::= & \mathit{method}\ \langle \mathit{methods} \rangle
\end{array}
$$

This defines a class called *cname*, implementing the specified interfaces. The optional *scname* gives the name of the direct superclass; if it is not present, the superclass is taken to be the root of the class hierarchy, namely `java.lang.Object`. The class *cname* inherits the methods and values present in its superclass, and these may be referred to in its definition.

As well as a superclass, a class can declare that it implements one or more interfaces. These correspond directly to the Java notion of an interface. Java libraries often require the creation of a class implementing a particular interface—for example, to use a Swing GUI one must create classes implementing various interfaces to be used as callbacks. Note that at the current time it is not possible to define interfaces in Camelot; they are provided purely for the purpose of interoperability.

Now we describe field declarations.

$$
\mathit{field} \quad ::= \quad \texttt{field}\ x : \tau \mid \texttt{field}\ \texttt{mutable}\ x : \tau \mid \texttt{val}\ x : \tau
$$

Instance fields are defined using the keyword `field`, and can optionally be declared to be mutable. Static fields are defined using `val`, and are non-mutable. In a sense these mutable fields are the first introduction of side-effects into Camelot. While the Camelot language is defined to have an array type, this has largely been ignored in our more formal treatments as it is not fundamental to the language. Mutable fields, on the other hand, are fundamental to our notion of object

orientation, so we expect any extension of Camelot resource-control features to O'Camelot to have to deal with this properly.

Methods are defined as follows, where $1 \leq i_1 \ldots i_m \leq n$.

$$
\begin{aligned}
\mathit{method} \quad ::= \quad & \texttt{maker(}x_1\texttt{:}\tau_1\texttt{)} \ldots \texttt{(}x_n\texttt{:}\tau_n\texttt{)} \ \langle \texttt{: super}\, x_{i_1} \ldots x_{i_m} \rangle = \mathit{exp} \\
& |\, \texttt{method}\, m\texttt{(}x_1\texttt{:}\tau_1\texttt{)} \ldots \texttt{(}x_n\texttt{:}\tau_n\texttt{)}\ \texttt{:}\ \tau = \mathit{exp} \\
& |\, \texttt{method}\, m\texttt{()}\ \texttt{:}\ \tau = \mathit{exp} \\
& |\, \texttt{let}\, m\texttt{(}x_1\texttt{:}\tau_1\texttt{)} \ldots \texttt{(}x_n\texttt{:}\tau_n\texttt{)}\ \texttt{:}\ \tau = \mathit{exp} \\
& |\, \texttt{let}\, m\texttt{()}\ \texttt{:}\ \tau = \mathit{exp}
\end{aligned}
$$

Again, we use the usual `let` syntax to declare what Java would call static methods. Static methods are simply *monomorphic* Camelot functions which happen to be defined within a class, although they are invoked using the syntax described earlier. Instance methods, on the other hand, are actually a fundamentally new addition to the language. We consider the instance methods of a class to be a set of mutually recursive monomorphic functions, in which the special variable `this` is bound to the current object of that class.

We can consider the methods as mutually recursive without using any additional syntax (such as `and` blocks) since they are monomorphic. ML uses `and` blocks to group mutually recursive functions because its *let-polymorphism* prevents any of these functions being used polymorphically in the body of the others, but this is not an issue here. In any case, this implicit mutual recursion feels appropriate when we are compiling to the Java Virtual Machine and have to come to terms with open recursion.

In addition to static and instance methods, we also allow a special kind of method called a *maker*. This is just what would be called a constructor in the Java world, but as in [4] we use the term maker in order to avoid confusion between object and datatype constructors. The `maker` term above defines a maker of the containing class C such that if `new` C is invoked with arguments of type $\tau_1 \ldots \tau_n$, an object of class C is created, the superclass maker is executed (this is the zero-argument maker of the superclass if none is explicitly specified), expression *exp* (of `unit` type) is executed, and the object is returned as the result of the `new` expression. Every class has at least one maker; a class with no explicit maker is taken to have the maker with no arguments which invokes the superclass zero-argument maker and does nothing. This implicit maker is inserted by the compiler.

4.4 TYPING

Typing rules for some of the more important Object Oriented extensions are given in Fig. 4.1. Rules for static method invocation and static field access are similar to those given for instance versions, and rules for the base language are roughly as one might expect, except that the rule for function application forces functions to be fully applied. The requirement above to state the types of fields, methods and makers at the point of definition means we can easily construct the sets of these types as makers(C), methods(C) and fields(C) for each class C.

$$\textsc{New}\ \frac{(\tau_1 \to \ldots \to \tau_n) \in \text{makers}(C) \quad \Sigma \vdash x_i : \tau_i' \quad \tau_i' \leqslant \tau_i}{\Sigma \vdash \texttt{new}\, C\, x_1 \ldots x_n : C}$$

$$\textsc{Invoke}\ \frac{\Sigma \vdash e : C \quad (id : \tau_1 \to \ldots \to \tau_n \to \tau) \in \text{methods}(C) \quad \Sigma \vdash x_i : \tau_i' \quad \tau_i' \leqslant \tau_i}{\Sigma \vdash e\#id\, x_1 \ldots x_n : \tau}$$

$$\textsc{Field}\ \frac{\Sigma \vdash e : C \quad (id : \tau) \in \text{fields}(C)}{\Sigma \vdash e\#id : \tau}$$

$$\textsc{Update}\ \frac{\Sigma \vdash e : C \quad (id : \tau) \in \text{fields}(C) \quad \Sigma \vdash e' : \tau}{\Sigma \vdash e\#id \texttt{<-}\, e' : \texttt{unit}}$$

$$\textsc{Cast}\ \frac{\Sigma \vdash e : \tau \quad \tau \leqslant \tau'}{\Sigma \vdash e :> \tau' : \tau'}$$

FIGURE 4.1 Additional Camelot typing rules

Consider rules INVOKE, and FIELD. Firstly, types must match exactly for field access, whereas methods can be called with subtypes of their argument types. Otherwise these are fairly similar.

Secondly, note that we look up methods(C) (respectively fields(C)). This implies that at the time this rule is applied the class C of the object in question must be known, at least in the obvious implementation. This has real consequences for the programmer—the programmer must ensure that the type of the object is suitably constrained at the time of invocation. In practice, this will probably mean that almost all function arguments of object type must be constrained before use and coercions may also be necessary in some places.

Additionally, method (and maker) overloading introduces ambiguity. Different instances of INVOKE or NEW may apply depending on the argument types, and indeed for many argument types there is no unique applicable method. In Java this is resolved by choosing the "most specific" method if it exists. In combination with the standard type inference algorithm this forces us to know the type of all arguments to a method at the point it is applied. Indeed in our current implementation this is exactly what happens; we assume argument types are available at the point of application and compute the most specific of the applicable methods. Again this puts a burden on the programmer, although in practice this has been proved in reasonable examples.

A more intelligent solution would only place constraints to be solved globally, but unfortunately these cannot be equality constraints, and so we have to depart from the simple unification algorithm. We are not alone in this problem; for example, the MLj implementation described in [2] also suffers from this. In [10], a new type inference algorithm is given for MLj which solves a system of more complex constraints using branching search and backtracking. Branching search is required because of the complexities of the type system, including implicit coercions such as `option`, and it may be that our more naive type system could use a simpler algorithm.

One way of avoiding these issues could be to avoid considering method invocations during type inference. Constraints could be inferred and solved by unification as usual, but with no constraints present for these invocations. After unification has taken place, we will be left with a typed program with some free type variables, and we can then resolve overloading in a more simplistic fashion (as the types of objects and method arguments should be known by this point). The remaining type variables will thus be instantiated after unification. Unfortunately this resolution requires another full typechecking, and indeed in our present implementation it may be easier to implement a system in the style of [10] if necessary.

Polymorphism

We remarked earlier that static methods are basically monomorphic Camelot functions together with a form of encapsulation, but it is worth considering polymorphism more explicitly. O'Camelot methods, whether static or instance methods, are not polymorphic. That is, they have subtype polymorphism but not parametric polymorphism (genericity), unlike Camelot functions which have parametric but not subtype polymorphism. This is not generally a problem, as most polymorphic functions will involve manipulation of polymorphic datatypes and can be placed in the main program, whereas most methods will be interfacing with the Java world and thus should conform to Java's subtyping polymorphism.

4.5 TRANSLATION

As mentioned earlier, the target for the present Camelot compiler is Java bytecode. However we make use of the intermediate language Grail (see [3]). Grail is a low-level functional language and is basically a functional form for Java bytecode. Grail's functional nature makes the compilation from Camelot more straightforward, but Grail is faithful enough to JVML that the compilation process is reversible.

Here we use the notation of Grail to describe the compilation of new Camelot features, but mostly the meanings of Grail phrases should be self-evident. However, it is worthwhile noting that the JVML basic blocks comprising a Camelot method are represented in Grail by a collection of mutually tail-recursive functions—calling these functions corresponds to JVML goto instructions. There are several different method invocation instructions, namely `invokestatic` for static methods, `invokevirtual` for instance methods, and `invokespecial` for calling object constructors—standard Camelot functions are tranlsated to static methods, and their application correpsonds to an `invokestatic` instruction. Grail differs from JVML by combining object creation and initialisation into the `new` instruction, but we must still use the `invokespecial` instruction to call the superclass constructor.

Notational issues aside, translating the new features is relatively straightforward, as the JVM (and Grail) provide what we need. In particular, Grail is suf-

```
fun β1(...)  =                          fun βn-1(...)  =
let                                     let
   val i = instance C1 ve                  val i = instance Cn-1 ve
in                                ...   in
   if i = 1 then γ1(...)                   if i = 1 then γn-1(...)
          else β2(...)                            else γn(...)
end                                     end

fun γ1 (...) =                    ...   fun γn (...) =
let                                     let
   val o1 = checkcast C1 ve                val on = checkcast Cn ve
in ρ1(...) end                          in ρn(...) end
```

FIGURE 4.2 Functions generated for `match` expression

ficiently expressive that it was not necessary to extend the compiler backend significantly.

Function ϕ below informally specifies the translation of the new Camelot expressions to Grail code. We assume these expressions are normalised in the style of the basic Camelot expressions, so that all subexpressions are atomic and have a simple Grail expansion, rather than requiring the generation of extra Grail functions and let statements.

$\phi(package.Class.method\ x_1 \ldots x_n) =$
$\quad$ `invokestatic` <τ_{ret} $package.Class.method$ τ_{arg}> `(`$\phi(x_1),\ldots,\phi(x_n)$`)`
$\phi(package.Class.field) =$ `getstatic` <τ $package.Class.field$>
$\phi($`new` $package.Class\ x_1 \ldots x_n) =$ `new` <$package.Class(\tau_{arg})$> `(`$\phi(x_1)\ldots\phi(x_n)$`)`
$\phi(obj\#mname\ x_1 \ldots x_n) =$
$\quad$ `invokevirtual` obj <τ_{ret} $package.Class.mname$ (τ_{arg})> `(`$\phi(x_1)\ldots\phi(x_n)$`)`
$\phi(obj\#field) =$ `getfield` obj <τ $package.Class.field$>
$\phi(obj\#field$<-$exp) =$ `putfield` obj <τ $package.Class.field$> exp
$\phi(obj$ `:>` $package.Class) =$ `checkcast` $package.Class\ obj$
$\phi($`isnull` $exp) =$ exp = `null[τ]`

Types τ, τ_{arg} and τ_{ret} are Grail types derived from the Camelot types inferred for the appropriate fields and methods. To illustrate the above translation, we show the translation of the multiplication of two `BigInteger` objects using the `multiply` instance method.

```
φ(n#multiply r) =
   invokevirtual n <java.math.BigInteger
   java.math.BigInteger.multiply
   (java.math.BigInteger)> (r)
```

The new match expressions are more complex. An example of the new type

of match statement is

```
match e with
        o_1 :> C_1 -> e_1
        ⋮
        o_n :> C_n -> e_n
```

where each C_i is a class name. We generate functions as in Fig 4.2, where β_1 will be the first function to be executed, i is a fresh variable, and v_e is a variable holding the result of evaluating expression e. Additionally we generate functions $\rho_1 \ldots \rho_n$ which compute the expressions $e_1 \ldots e_n$ then proceed with the current computation.

Making Classes

Translating class definitions is fairly straightforward. A `val` declaration corresponds to a final static field, the type of which is the translation of the stated Camelot type. Similarly a `field` definition corresponds to an instance field of the appropriate type, which will be `final` if the field is not `mutable`.

A `maker` corresponds to a method called `<init>` taking arguments of the appropriate type (returning `void`), and calling the appropriate `<init>` method in the superclass before executing the code corresponding to expression in the body, which is compiled as above.

As remarked earlier, static methods are basically monomorphic Camelot functions encapsulated in a class, and so their compilation is just as standard Camelot functions. Instance methods are also compiled like monomorphic Camelot functions, but references to `this` are permitted.

4.6 OBJECTS AND RESOURCE TYPES

As described in Sec. 4.2, the use of diamond annotations on Camelot programs in combination with certain resource-aware type systems allows the heap usage of those programs to be inferred, as well as allowing some in-place update to occur. Clearly the presence of mutable objects in O'Camelot also provides for in-place update. However by allowing arbitrary object creation we also replicate the unbounded heap-usage problem solved for datatypes. Perhaps more seriously, we are allowing Camelot programs to invoke arbitrary Java code, which may use an unlimited amount of heap space.

First consider the second problem. Even if we have some way to place a bound on the heap space used by our new OO features within a Camelot program, external Java code may use any amount of heap whatsoever. There seem to be a few possible approaches to this problem, none of which are particularly satisfactory. We could decide only to allow the use of external classes if they came with a proof of bounded heap usage. Constructing a resource-bounded Java class library or inferring resource bounds for an existing library would be a massive undertak-

ing, although perhaps less problematic with the smaller class libraries used with mobile devices. This suggestion seems somewhat unrealistic.

Alternatively, we could simply allow the resource usage of external methods to be stated by the programmer or library creator. This extends the trusted computing base in the sense of resources, but seems a more reasonable solution. The other alternative—considering resource-bound proofs only to refer to the resources directly consumed by the Camelot code—seems unrealistic, as one could easily (and even accidentally) cheat by using Java libraries to do some memory-consuming "dirty work".

The issue of heap-usage *internal* to O'Camelot programs seems more tractable, although we do not propose a solution here. A first attempt might mimic the techniques used earlier for datatypes; perhaps we can adapt the use of diamonds and linear type systems? The use of diamonds for in-place update is irrelevant here and indeed relies on the uniform representation of datatypes by objects of a particular Java class. Since we are hardly going to represent every Java object by an object of one class we could not hope to have such a direct correlation between diamonds and chunks of storage.

However, we could imagine an abstract diamond which represents the heap storage used by an arbitrary object and require any instance of `new` to supply one of these diamonds, in order that the total number of objects created is limited. Unfortunately reclamation of such an abstract diamond would only correspond to making an object available to garbage collection, rather than definitely being able to re-use the storage. Even so, such a system might be able to give a measure of the total number of objects created and the maximum number in active use simultaneously.

4.7 RELATED WORK

We have made reference to MLj, the aspects of which related to Java interoperability are described in [2]. MLj is a fully formed implementation of Standard ML and as such is a much larger language than we consider here. In particular, MLj can draw upon features from SML such as modules and functors, for example, allowing the creation of classes parameterised on types. Such flexibility comes with a price, and we hope that the restrictions of our system will make the certification of the resource usage of O'Camelot programs more feasible.

By virtue of compiling an ML-like language to the JVM, we have made many of the same choices that have been made with MLj. In many cases there is one obvious translation from high level concept to implementation, and in others the appropriate language construct is suggested by the Java object system. However, we have also made different choices more appropriate to our purpose, in terms of transparency of resource usage and wanting a smaller language. For example, we represent objects as records of mutable fields whereas MLj uses immutable fields holding references.

There have been various other attempts to add object-oriented features to ML and ML-like languages. O'Caml provides a clean, flexible object system with

many features and impressive type inference—a formalised subset is described in [12]. As in O'Camelot, objects are modelled as records of mutable fields plus a collection of methods. Many of the additional features of O'Caml could be added to O'Camelot if desired, but there are some complications caused when we consider Java compatibility. For example, there are various ways to compile parameterised classes and polymorphic methods for the JVM, but making these features interact cleanly with the Java world is more subtle.

The power of the O'Caml object system seems to come more from the distinctive type system employed. O'Caml uses the notion of a *row variable*, a type variable standing for the types of a number of methods. This makes it possible to express "a class with these methods, and possibly more" as a type. Where we would have a method parameter taking a particular object type and by subsumption any subtype, in O'Caml the type of that parameter would include a row variable, so that any object with the appropriate methods and fields could be used. This allows O'Caml to preserve type inference, but this is less important for our application and does not map cleanly to the JVM.

A class mechanism for Moby is defined in [4] with the principle that classes and modules should be orthogonal concepts. Lacking a module system, Camelot is unable to take such an approach, but both Moby and O'Caml have been a guide to concrete representation. Many other relevant issues are discussed in [9], but again lack of a module system—and our desire to avoid this to keep the language small—gives us a different perspective on the issues.

4.8 CONCLUSION

We have described the language Camelot and its unique features enabling the control of heap-allocated data and have outlined an object-oriented extension allowing interoperability with Java programs and libraries. We have kept the language extension fairly minimal in order to facilitate further research on resource aware programming, yet it is fully-featured enough for the mobile applications we envisage for Camelot.

The O'Camelot compiler implements all the features described here. The current version of the compiler can be obtained from

`http://www.lfcs.inf.ed.ac.uk/mrg/camelot/`

A EXAMPLE

Here we give an example of the features defined above. The code below, together with the two standard utility functions `rev` and `len` for list reversal and length, defines a program for Sun's MIDP platform (as described in [11]), which runs on devices such as PalmOS PDAs. The program displays the list of primes in an interval. Two numbers are entered into the first page of the GUI, and when a button is pressed a second screen appears with the list of primes, calculated using the sieve of Eratosthenes, along with a button leading back to the initial display.

This example has been compiled with our current compiler implementation, and executed on a PalmOS device.

```
class primes = javax.microedition.midlet.MIDlet with
  implement javax.microedition.lcdui.CommandListener

  field exitCommand: javax.microedition.lcdui.Command
  field goCommand: javax.microedition.lcdui.Command
  field doneCommand: javax.microedition.lcdui.Command
  field mainForm: javax.microedition.lcdui.Form
  (* lower and upper limits: *)
  field lltf: javax.microedition.lcdui.TextField
  field ultf: javax.microedition.lcdui.TextField
  field display: javax.microedition.lcdui.Display

  maker () =
    let _ = display <-
      (javax.microedition.lcdui.Display.getDisplay
        (this:> javax.microedition.midlet.MIDlet))
    in let _ = goCommand <-
      (new javax.microedition.lcdui.Command
      "Go" javax.microedition.lcdui.Command.SCREEN 1)
    in let _ = exitCommand <-
      (new javax.microedition.lcdui.Command
      "Exit" javax.microedition.lcdui.Command.SCREEN  2)
    in let t = new javax.microedition.lcdui.Form "Primes"
    in let ll = new javax.microedition.lcdui.TextField
        "Lower limit:" "" 10
        javax.microedition.lcdui.TextField.NUMERIC
    in let _ = lltf <- ll
    in let _ = t#append ll
    in let ul = new javax.microedition.lcdui.TextField
        "Upper limit:" "" 10
        javax.microedition.lcdui.TextField.NUMERIC
    in let _ = ultf <- ul
    in let _ = t#append ul
    in let _ = t#addCommand (this#goCommand)
    in let _ = t#addCommand (this#exitCommand)
    in let _ = mainForm <- t
    in t#setCommandListener this

  method startApp (): unit =
    this#display#setCurrent (this#mainForm)
  method pauseApp (): unit = ()
  method destroyApp (b:bool): unit = ()
  method commandAction
           (cmd: javax.microedition.lcdui.Command)
           (s: javax.microedition.lcdui.Displayable)
           : unit =
```

```
    if cmd#equals (this#exitCommand)
    then let _ = this#destroyApp false
         in this#notifyDestroyed ()
    (* create & display list of primes *)
    else if cmd#equals (this#goCommand)
    then
      let    lower_limit = int_of_string
                           (this#lltf#getString())
      in let upper_limit = int_of_string
                           (this#ultf#getString())
      in let primes =
               new javax.microedition.lcdui.Form "Primes"
      in let _ = appendPrimes lower_limit upper_limit primes
      in let done = new javax.microedition.lcdui.Command
                "Done"
                javax.microedition.lcdui.Command.SCREEN 1
      in let _ = doneCommand <- done
      in let _ = primes#addCommand done
      in let _ = primes#setCommandListener this
      in let _ =
        javax.microedition.lcdui.AlertType.INFO#playSound
          (this#display)
      in this#display#setCurrent primes
   (* back to main form *)
    else if cmd#equals (this#doneCommand) then
      this#display#setCurrent (this#mainForm)
    else ()
end
(* Generate a list of prime numbers in an interval [a..b] *)
(* Integer square roots *)
let increase k n = if (k+1)*(k+1) > n then k else k+1
let rec intsqrt n = if n = 0 then 0
                    else increase (2*(intsqrt (n/4))) n

(* n is divisible by no member of l which is <= sqrt n *)
let isPrime n l lim =
  match l with
    [] -> true
  | h::t -> h <= lim && n mod h <> 0 && isPrime n t lim

(* generate list of primes between n and top *)
let make1 n top acc =
  if n > top then rev acc []
  else if isPrime n acc n then make1 (n+2) top (n::acc)
  else make1 (n+2) top acc

let makeSmallPrimes top = make1 3 top [2]
let makePrimes n top smallPrimes =
  if n > top then []
```

```
  else if isPrime n smallPrimes n then
    n::(makePrimes (n+2) top smallPrimes)
  else makePrimes (n+2) top smallPrimes

let appList l (f: javax.microedition.lcdui.Form) =
  match l with [] -> ()
  | (h::t)@_ -> let _ = f#append ( (string_of_int h)^"\n")
                in appList t f

let appendPrimes bot top
      (f: javax.microedition.lcdui.Form) =
  let smallPrimes = makeSmallPrimes (intsqrt top)
  in let primes = makePrimes (bot + 1 - bot mod 2)
                      top smallPrimes
  in let s = (string_of_int (len primes)) ^ " primes\n"
  in let _ = f#append s
  in appList primes f
```

REFERENCES

[1] O'Caml. See `http://www.ocaml.org`.

[2] Nick Benton and Andrew Kennedy. Interlanguage working without tears: Blending SML with Java. In *Proc. of ICFP*, pages 126–137, 1999.

[3] Lennart Beringer, Kenneth MacKenzie, and Ian Stark. Grail: a functional form for imperative mobile code. In Vladimiro Sassone, editor, *Electronic Notes in Theoretical Computer Science*, volume 85. Elsevier, 2003.

[4] K. Fisher and J. Reppy. Moby objects and classes, 1998. Unpublished manuscript.

[5] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[6] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *POPL'03*, January 2003.

[7] Michal Konečný. Typing with conditions and guarantees in LFPL. In *Types for Proofs and Programs: Proceedings of the International Workshop TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2002.

[8] K. MacKenzie and N. Wolverson. Camelot and Grail: Resource-aware functional programming for the JVM. In *Trends in Functional Programming Volume 4: Proceedings of TFP2003*, pages 29–46. Intellect, 2004.

[9] David MacQueen. Should ML be object-oriented? *Formal Aspects of Computing*, 13(3-5), 2002.

[10] Bruce McAdam. Type inference for MLj. In Stephen Gilmore, editor, *Trends in Functional Programming*, volume 2. Intellect, 2000.

[11] Sun Microsystems. Mobile Information Device Profile (MIDP). See `http://java.sun.com/products/midp/`.

[12] Didier Remy and Jerome Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

Chapter 5

Static Single Information from a Functional Perspective

Jeremy Singer[1]

Abstract Static single information form is a natural extension of the well-known static single assignment form. It is a program intermediate representation used in optimising compilers for imperative programming languages. In this paper we show how a program expressed in static single information form can be transformed into an equivalent program in functional notation. We also examine the implications of this transformation.

5.1 INTRODUCTION

Static single information form (SSI) [2] is a natural extension of the well-known static single assignment form (SSA) [11]. SSA is a compiler intermediate representation for imperative programs that enables precise and efficient analyses and optimisations.

In SSA, each program variable has a unique definition point. To achieve this, it is necessary to rename variables and insert extra pseudo-definitions (ϕ-functions) at control flow merge points. Control flow merge points occur at the start of basic blocks. A basic block is a (not necessarily maximal) sequence of primitive instructions with the property that if control reaches the first instruction, then all instructions in the basic block will be executed. SSA programs have the desirable property of referential transparency—that is, the value of an expression depends only on the value of its subexpressions and not on the order of evaluation or side-effects of other expressions. Referentially transparent programs are easier to analyse and reason about.

We take the following simple program as an example:

[1]University of Cambridge Computer Laboratory,
William Gates Building, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
Email: `jeremy.singer@cl.cam.ac.uk`

```
1:  z ← input()
2:  if (z = 0)
3:        then y ← 42
4:        else y ← z + 1
5:  output(y)
```

To convert this program into SSA form, we have to rename instances of variable y so that each new variable has only a single definition point in the program.

The SSA version of the program is shown below:

```
1:  z ← input()
2:  if (z = 0)
3:        then y0 ← 42
4:        else y1 ← z + 1
5:  y2 ← φ(y0, y1)
6:  output(y2)
```

The ϕ-function merges (or multiplexes) the two incoming definitions of y_0 and y_1 at line 5. If the path of execution comes from the `then` branch, then the ϕ-function takes the value of y_0, whereas if the path of execution comes from the `else` branch, then the ϕ-function takes the value of y_1.

SSI is an extension of SSA. It introduces another pseudo-definition, the σ-function. When converting to SSI, in addition to renaming variables, and inserting ϕ-functions at control flow merge points, it is necessary to insert σ-functions at control flow split points. We have contrasted SSA and SSI at length elsewhere [25], in terms of their computation and data flow analysis properties. It is sufficient to say that SSI can be computed almost as efficiently as SSA and that SSI permits a wider range of data flow analysis techniques than SSA.

The σ-function is the exact opposite of the ϕ-function. The differences are tabulated in Fig. 5.1.

We now convert the above program into SSI:

```
1:  z0 ← input()
2:  if (z0 = 0)
3:  z1, z2 ← σ(z0)
4:        then y0 ← 42
5:        else y1 ← z2 + 1
6:  y2 ← φ(y0, y1)
7:  output(y2)
```

The σ-function splits (or demultiplexes) the outgoing definition of z_0 at line 3. If the path of execution proceeds to the `then` branch, then the σ-function assigns the value of z_0 to z_1. However, if the path of execution proceeds to the `else` branch, then the σ-function assigns the value of z_0 to z_2.

Since SSI is such a straightforward extension of SSA, it follows that algorithms for SSA can be quickly and naturally modified to handle SSI. For example,

ϕ-function	σ-function
inserted at control flow merge points	inserted at control flow split points
placed at start of basic block	placed at end of basic block
single destination operand	n destination operands, where n is the number of successors to the basic block that contains this σ-function
n source operands, where n is the number of predecessors to the basic block that contains this ϕ-function.	single source operand
takes the value of one of its source operands (dependent on control flow) and assigns this value to the destination operand	takes the value of its source operand and assigns this value to one of the destination operands (dependent on control flow)

FIGURE 5.1 Differences between ϕ- and σ-functions

the standard SSA computation algorithm [11] can be simply extended to compute SSI instead [23]. Similarly, the SSA conditional constant propagation algorithm [29] has a natural analogue in SSI [2], which produces even better results.

It is a well-known fact that SSA can be seen as a form of functional programming [6]. Inside every SSA program, there is a functional program waiting to be released. Therefore, we should not be surprised to discover that SSI can also be seen as a form of functional programming.

Consider the following program, which calculates the factorial of 5.

1: $r \leftarrow 1$
2: $x \leftarrow 5$
3: **while** $(x > 0)$ **do**
4: $\quad r \leftarrow r * x$
5: $\quad x \leftarrow x - 1$
6: **done**
7: **return** r

First we convert this program into a standard control flow graph (CFG) [1], as shown in Fig. 5.2. Then we translate this program into SSI form, as shown in Fig. 5.3. This SSI program can be simply transformed into the functional program shown in Fig. 5.4.

In the conversion from SSA to functional notation, a basic block #n that begins with one or more ϕ-functions is transformed into a function f_n. Jumps to such basic blocks become tail calls to the corresponding functions. The actual parameters of the tail calls are the source operands of the ϕ-functions. The formal parameters of the corresponding functions are the destination operands of the

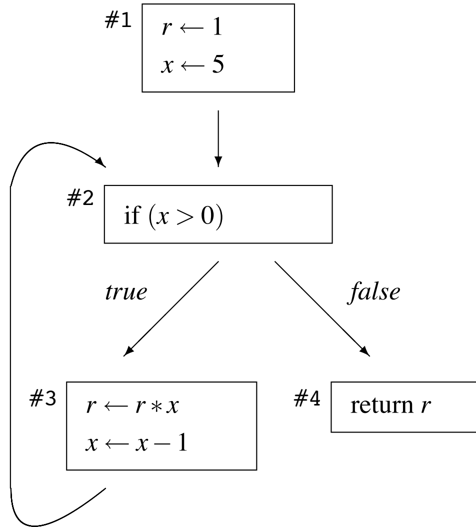

FIGURE 5.2 Control flow graph for factorial program

ϕ-functions.

In the conversion from SSI to functional notation, in addition to the above transformation, whenever a basic block ends with one or more σ-functions, then successor blocks #p and #q are transformed into functions f_p and f_q. Jumps to such successor blocks become tail calls to the corresponding functions. The actual parameters of the tail calls are the source operands of the σ-functions. The formal parameters of the corresponding functions are the relevant destination operands of the σ-functions. (We notice again that σ-functions have analogous properties to ϕ-functions.)

The main technical contribution of this paper is the detailed presentation of an algorithm to convert SSI programs into a functional intermediate representation. The remainder of the paper is laid out as follows: in section 5.2 we review the previous work in this area, in section 5.3 we formally define SSI, in section 5.4 we present the algorithm to transform SSI code into a functional program, in section 5.5 we show how there are both an optimistic and a pessimistic version of this transformation, in section 5.6 we contemplate the possibility of recovering SSI from a functional program (the reverse transformation), in section 5.7 we discuss why the transformation from SSI to functional notation may be useful, then finally in section 5.8 we draw some conclusions.

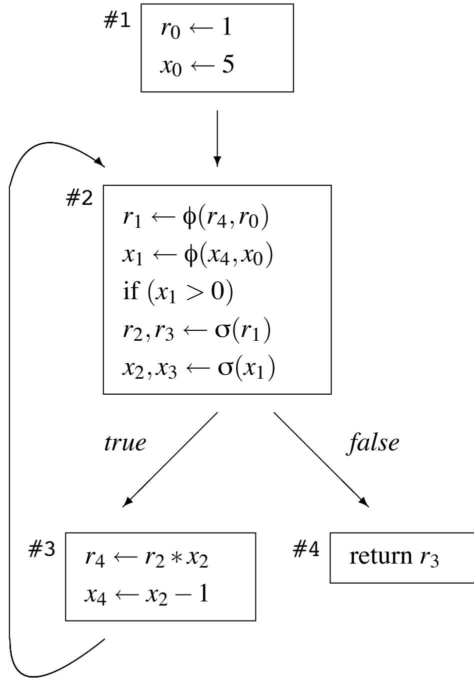

FIGURE 5.3 Static single information form for factorial program

5.2 RELATED WORK

To the best of our knowledge no-one has attempted to transform SSI into a functional notation. Ananian [2] gives an executable representation for SSI, but this is defined in terms of demand-driven operational semantics and seems rather complicated.

Several people have noted a correspondence between programs in SSA and λ-calculus. Kelsey [16] shows how to convert continuation passing style [4] into SSA and vice versa. Appel [6] informally shows the correspondence between SSA and functional programming. He gives an algorithm [5] for translating SSA to functional intermediate representation. (We extend Appel's algorithm in section 5.4 of this paper.)

Chakravarty et al. [10] formalise a mapping from programs in SSA form to administrative normal form (ANF) [12]. ANF is a restricted form of λ-calculus. They also show how the standard SSA conditional constant propagation algorithm [29] can be rephrased in terms of ANF programs.

```
let r0 = 1, x0 = 5
in
  let function f2(r1,x1) =
    let function f3(r2,x2) =
      let r4 = r2 * x2, x4 = x2 - 1
      in
        f2(r4,x4)
    and function f4(r3,x3) =
      return r3
    in
      if (x1 > 0)
        then f3(r1,x1)
        else f4(r1,x1)
  in
    f2(r0,x0)
```

FIGURE 5.4 Functional representation for SSI factorial program

5.3 STATIC SINGLE INFORMATION

Static single information form (SSI) was originally described by Ananian [2]. He states that "the principal benefits of using SSI form are the ability to do predicated and backwards data flow analyses efficiently". He gives several examples including very busy expressions analysis and sparse predicated typed constant propagation. Indeed, SSI has been applied to a wide range of problems [22, 28, 14, 3].

The MIT Flex compiler [13] uses SSI as its intermediate representation. Flex is a compiler for Java, written in Java. As far as we are aware, Flex is the only publicly available SSI-based compiler. However, we are adding support for SSI to Machine SUIF [27], an extensible compiler infrastructure for imperative languages like C and Fortran. We have implemented an efficient algorithm for SSI computation [23] and several new SSI analysis passes.

Below, we give the complete formal definition of a transformation from CFG to SSI notation. This definition is taken from Ananian [2]. A few auxiliary definitions may be required before we quote Ananian's SSI definition. The original program is the classical CFG representation of the program [1]. Program statements are contained within nodes (also known as basic blocks). Directed edges between nodes represent the possible flow of control. A path is a sequence of consecutive edges. $\rightarrow^{+}$ represents a path consisting of at least one edge (a non-null path). There is a path from the START node to every node in the CFG and there is a path from every node in the CFG to the END node. The new program is in SSI. It is also a CFG, but it contains additional pseudo-definition functions and the variables have been renamed. The variables in the original program are referred to as the original variables. The SSI variables in the new program are

referred to as the new variables.

So, here is Ananian's definition:

1. If two nonnull paths $X \to^+ Z$ and $Y \to^+ Z$ exist having only the node Z where they converge in common, and nodes X and Y contain either assignments to a variable V in the original program or a ϕ- or σ-function for V in the new program, then a ϕ-function for V has been inserted at Z in the new program. (Placement of ϕ-functions)

2. If two nonnull paths $Z \to^+ X$ and $Z \to^+ Y$ exist having only the node Z where they diverge in common, and nodes X and Y contain either uses of a variable V in the original program or a ϕ- or σ-function for V in the new program, then a σ-function for V has been inserted at Z in the new program. (Placement of σ-functions)

3. For every node X containing a definition of a variable V in the new program and node Y containing a use of that variable, there exists at least one path $X \to^+ Y$ and no such path contains a definition of V other than at X. (Naming after ϕ-functions)

4. For every pair of nodes X and Y containing uses of a variable defined at node Z in the new program, either every path $Z \to^+ X$ must contain Y or every path $Z \to^+ Y$ must contain X. (Naming after σ-functions)

5. For the purposes of this definition, the START node is assumed to contain a definition and the END node a use for every variable in the original program. (Boundary conditions)

6. Along any possible control flow path in a program being executed consider any use of a variable V in the original program and the corresponding use V_i in the new program. Then, at every occurrence of the use on the path, V and V_i have the same value. The path need not be cycle-free. (Correctness)

Ananian's original SSI computation algorithm can be performed in $O(EV)$ time, where E is a measure of the number of edges in the control flow graph and V is a measure of the number of variables in the original program. This is worst case complexity, but typical time complexity is linear in the program size.

5.4 TRANSFORMATION

In this section we present the algorithm that transforms SSI into a functional notation.

We adopt a cut-down version of Appel's functional intermediate representation (FIR) [5]. The abstract syntax of our FIR is given in Fig. 5.5. FIR has the same expressive power as ANF [12]. Expressions are broken down into primitive operations whose order of evaluation is specified. Every intermediate result is an explicitly named temporary. Every argument of an operator or function is an

atom	$\rightarrow c$	constant integer
atom	$\rightarrow v$	variable
exp	$\rightarrow$ **let** *fundefs* **in** *exp*	function declaration
exp	$\rightarrow$ **let** $\underline{v}$ = *atom* **in** *exp*	copy
exp	$\rightarrow$ **let** $\underline{v}$ = *binop*(*atom*,*atom*) **in** *exp*	arithmetic operator
exp	$\rightarrow$ **if** *atom relop atom* **then** *exp* **else** *exp*	conditional branch
exp	$\rightarrow$ *atom*(*args*)	tail call
exp	$\rightarrow$ **let** $\underline{v}$ = *atom*(*args*) **in** *exp*	non-tail call
exp	$\rightarrow$ **return** *atom*	return
args	$\rightarrow$	
args	$\rightarrow$ *atom args*	
fundefs	$\rightarrow$	
fundefs	$\rightarrow$ *fundefs* **function** $\underline{v}$(*formals*) = *exp*	
formals	$\rightarrow$	
formals	$\rightarrow$ $\underline{v}$ *formals*	
binop	$\rightarrow$ **plus** | **minus** | **mul** | ...	
relop	$\rightarrow$ **eq** | **ne** | **lt** | ...	

FIGURE 5.5 Functional intermediate representation

atom (variable or constant). As in SSA, SSI and λ-calculus, every variable has a single assignment (binding), and every use of that variable is within the scope of the binding. (In Fig. 5.5, binding occurrences of variables are underlined.) No variable name can be used in more than one binding. Every binding of a variable has a scope within which all the uses of that variable must occur.

- For a variable bound by **let** $v = \ldots$ **in** *exp*, the scope of v is just *exp*.
- The scope of a function variable f_i bound in

  let function $f_1(\ldots) = exp_1 \ldots$
      function $f_k(\ldots) = exp_k$
  in *exp*

  includes all the exp_j (to allow for mutually recursive functions) as well as *exp*.
- For a variable bound as the formal parameter of a function, the scope is the body of that function.

Any SSI program can be translated into FIR. Each basic block with more than one predecessor is transformed into a function. The formal parameters of that

function are the destination operands of the ϕ-functions in that basic block. (If the block has no ϕ-functions then it is transformed into a parameterless function.) Similarly, each basic block that is the target of a conditional branch instruction is transformed into a function. The formal parameters of that function are the appropriate destination operands of the σ-functions in the preceding basic block (that is to say, the σ-functions that are associated with the conditional branch). We assume that the SSI program is in edge-split form—no basic block with multiple successors has an edge to a basic block with multiple predecessors. In particular this means that basic blocks that are the targets of a conditional branch can only have a single predecessor. (It should always be possible to transform an SSI program into edge-split form.)

If block f dominates block g, then the function for g will be nested inside the body of the function for f. Instead of jumping to a block which has been transformed into a function, a tail call replaces the jump. The actual parameters of the tail call will be the appropriate source operands of corresponding σ- or ϕ-functions. (Every conditional branch will dominate both its `then` and `else` blocks, in edge-split SSI.)

The algorithm for transforming SSI into FIR is given in Fig. 5.6. It is based on algorithm 19.20 from Appel's book [5]. `Translate()` ensures function definitions are correctly nested. `Statements()` outputs FIR code for each basic block. Appel's algorithm handles SSA, so we extend it to deal with SSI instead. In our algorithm lines of code that have been altered from Appel's original SSA-based algorithm are marked with a `!` and entirely new lines of code (to handle SSI-specific cases) are marked with a `+`. In the code for the `Statements()` function, $\oplus$ represents the general case for binary arithmetic operators and $<$ represents the general case for binary relational operators.

5.5 OPTIMISTIC VERSUS PESSIMISTIC

There are two different approaches to computing SSI. Ananian's approach [2] is pessimistic, in that it assumes that ϕ- and σ-functions are needed everywhere, then it removes such functions when it can show that they are not actually required. This is a kind of greatest fixed point calculation. (Aycock and Horspool adopt the same pessimistic approach in their generation of SSA [8].) The alternative approach to computing SSI [23] is optimistic. It assumes that no ϕ- or σ-functions are needed, then it inserts such functions when it can show that they are actually required. This is a kind of least fixed point calculation. (The classical SSA computation algorithm [11] employs the same optimistic approach.) Ananian claims that this optimistic approach ought to take longer, but in practice it seems to be faster than the pessimistic approach.

Just as there is an optimistic and a pessimistic approach to the computation of SSI, there appear to be an optimistic and a pessimistic approach to the transformation into functional notation. The pessimistic approach takes the original program CFG and converts each basic block into a top-level function, with tail calls to appropriate successor functions. Each generated top-level function has a

formal parameter for every program variable, and each function call site has an actual parameter for every program variable. Appel [6] refers to this as the "really crude approach." Useless parameters may be identified and eliminated with the help of liveness and other data flow information. The necessary parameters for each functional block should be those variables which are live at each corresponding basic block boundary in the original program. (A variable is live at a particular program point if there is a control flow path from that point along which the variable's value may be used before that variable is redefined.) This makes sense since SSI is an encoding of liveness information, as Ananian states [2].

The optimistic approach is exactly as given in section 5.4. It can be explained in the following manner. It uses the dominance relations of the control flow graph to determine how the functional blocks should be nested. (Nesting is required in order for functional blocks to use variables declared in outer scope.) Then it applies standard lambda lifting techniques [15] to generate the appropriate parameters for each functional block.

A formal clarification of the relationship between optimistic and pessimistic computation of SSI is the subject of ongoing research.

5.6 CONVERTING FUNCTIONAL PROGRAMS BACK TO SSI

It is possible to transform an arbitrary program p expressed in FIR into SSI, simply by treating p as an imperative program. (Let-bound atomic variables become mutable virtual registers and function applications become procedure calls.) Standard SSI computation techniques [2, 23] can then be applied to the imperative program.

However, suppose that a program p_{SSI} in SSI has been transformed into a program p_{func} in FIR. In this section we address the concept of recovering p_{SSI} from p_{func}.

p_{func} is in SSA, since each let-bound variable is only assigned a value at one program point. However p_{func} is not in SSI, since the same parameters are supplied to the tail calls on either side of an `if` statement. (Recall that these parameters correspond to the source parameters of the σ-functions associated with this conditional branch in p_{SSI}.) The simplest way to transform p_{func} into a valid SSI program, say p'_{SSI}, is to add σ-functions at each `if` statement, and rename the parameters of the tail calls accordingly. There is a drawback with this approach however. Now imagine converting p'_{SSI} into FIR using our algorithm. There would be an additional layer of function calls at the `if` statements, because of the extra σ-functions. Admittedly these extra function calls could be removed by limited β-contraction, but it is embarrassing to admit that converting from SSI to FIR and back to SSI (ad infinitum) does not reach a fixed point. In fact this is a diverging computation.

The problem is that the σ-functions are already encoded as function calls in p_{func} but we do not recover this information. We insert extra σ-functions instead. One way to avoid this would be to inline (β-contract) all functions in p_{func} that

are only called from one call site (this includes all functions that originated from σ-functions). If this transformation is done prior to the insertion of σ-functions, then the problem of an extra layer of function call indirection does not arise.

Kelsey [16] gives a method for recovering ϕ-functions from functional programs. We should be able to apply similar techniques to p_{func}. Thus it should be possible to recover (something resembling) p_{SSI} from p_{func}.

5.7 MOTIVATION

In this section we briefly consider why the transformation from SSI into functional notation may be of value.

Typed functional languages may be useful as compiler intermediate representations for imperative languages. There has recently been a great deal of research effort in this area, with systems such as typed assembly language [18], proof carrying code [20, 7] and the value dependence graph [30]. SSA and SSI fit neatly into this category, since they can be seen from a functional perspective, and they are most amenable to high-level type inference techniques [19, 26]. The implementors of similar typed functional representations for Java bytecode, such as λJVM [17] and GRAIL [9], comment that a functional representation makes both verification and analysis straightforward. It is useful for reasoning about program properties, such as security and resource consumption guarantees. Functional notations are also well-suited for translation into lower-level program representations.

It is certainly true that algorithms on such functional representations can often be more rigorously defined [10] and proved correct. It would be interesting to compare existing SSA or SSI data flow analyses with the equivalent analyses in the functional paradigm, perhaps to discover similarities and differences. Such cross-community experience is often instructive to one of the parties, if not both.

We have effectively made SSI interprocedural in scope, by abstracting all control flow into function calls. Until now, SSI has only been envisaged as an intraprocedural representation, and it has not been clear how to extend SSI to whole program scope. Now there is no longer any distinction between intraprocedural and interprocedural control flow.

Finally we note that the functional representation of SSI programs is executable. Standard SSI is not an executable representation; it is restricted in the same manner as original SSA. (ϕ- and σ-functions require some kind of runtime support to determine which value to assign to which variable.) Ananian has concocted an operational semantics for an extended version of SSI [2], however this is quite complex and unwieldy to use. On the other hand, functional programs are natural, understandable and easily executable with a well-known semantics. We have successfully translated some simple SSI programs into Haskell and ML code, using the transformation algorithm of section 5.4.

For instance, Fig. 5.7 shows the dynamic data flow graph [21] of three Haskell factorial functions that each compute 5! (the answer is 120). The three values are then added together (the sum total is 360). The left portion of the graph represents

a standard Haskell iterative definition of the factorial function:

```
faci 0 acc = acc
faci n acc = fac1 (n-1) (acc*n)
```

The middle portion of the graph represents a standard Haskell recursive definition of the factorial function:

```
facr 0 = 1
facr n = n * facr (n-1)
```

The right portion of the graph represents the Haskell version of the functional program from figure 5.4 which is the transformation of the SSI program from figure 5.3. We notice that the right portion of the dynamic data flow graph has exactly the same shape as the left portion, which reveals that both are computing factorials iteratively, so we see that the transformation from imperative to functional style does not alter the data flow behaviour of the program at all.

5.8 CONCLUSIONS

In this paper we have shown how SSI (generally regarded as an imperative program representation) can be converted into a simple functional notation. We have specified a transformation algorithm and we have briefly discussed the possible applications of this transformation process.

Compilers for functional programming languages (such as the Glasgow Haskell compiler) often translate their intermediate form into an imperative language (such as C), which is then compiled to machine code. This seems rather wasteful, since the C compiler (if it uses a functional representation as its intermediate form) will attempt to reconstruct the functional program which has been carelessly thrown away by the functional compiler backend.

Finally we comment on future work. The transformation algorithm presented in section 5.4 could possibly be formalised, in the same manner as Appel's original work on SSA [6, 5] has been formalised [10]. Next we need to translate existing SSI analysis algorithms to this new functional framework. We must also consider how to take advantage of this functional notation in order to devise new analyses and optimisations.

On a different note, SSA and SSI are just two members of a large family of renaming schemes [24]. It would be interesting to see if every scheme in the family could be converted to a functional notation, using the same general techniques outlined in this paper.

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[2] C. S. Ananian. The static single information form. Master's thesis, Massachusetts Institute of Technology, Sep 1999.

[3] C. S. Ananian and M. Rinard. Data size optimizations for Java programs. In *Proceedings of the 2003 ACM SIGPLAN conference on Languages, Compilers, and Tools for Embedded Systems*, pages 59–68, 2003.

[4] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[5] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, first edition, 1998.

[6] A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, Apr 1998.

[7] A. W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science*, pages 247–256, 2001.

[8] J. Aycock and N. Horspool. Simple generation of static single assignment form. In *Proceedings of the 9th International Conference in Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 110–125. Springer-Verlag, 2000.

[9] L. Beringer, K. MacKenzie, and I. Stark. Grail: a functional form for imperative mobile code. *Electronic Notes in Theoretical Computer Science*, 85(1), 2003.

[10] M. M. Chatravarty, G. Keller, and P. Zadarnowski. A functional perspective on SSA optimisation algorithms. In *Proceedings of the 2nd International Workshop on Compiler Optimization Meets Compiler Verification*, 2003. http://www.cse.unsw.edu.au/~patrykz/papers/ssa-lambda/.

[11] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[12] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 237–247, 1993.

[13] The Flex compiler infrastructure, 1998. http://www.flex-compiler.lcs.mit.edu/Harpoon/.

[14] O. Gheorghioiu, A. Salcianu, and M. Rinard. Interprocedural compatibility analysis for static object preallocation. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284, 2003.

[15] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[16] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, Mar 1995.

[17] C. League, V. Trifonov, and Z. Shao. Functional Java bytecode. In *Proceedings of the 5th World Conference on Systemics, Cybernetics, and Informatics—Workshop on Intermediate Representation Engineering for the Java Virtual Machine*, 2001. http://flint.cs.yale.edu/flint/publications/lamjvm.html.

[18] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[19] A. Mycroft. Type-based decompilation. In *Proceedings of the European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 208–223. Springer-Verlag, 1999.

[20] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[21] N. Nethercote and A. Mycroft. Redux: A dynamic dataflow tracer. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[22] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices and accessed memory regions. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 182–195, 2000.

[23] J. Singer. Efficiently computing the static single information form, 2002. http://www.cl.cam.ac.uk/˜jds31/research/computing.pdf.

[24] J. Singer. A framework for virtual register renaming schemes, 2003. http://www.cl.cam.ac.uk/˜jds31/research/renaming.pdf.

[25] J. Singer. SSI extends SSA. In *Work in Progress Session Proceedings of the Twelth International Conference on Parallel Architectures and Compilation Techniques*, 2003. http://www.cl.cam.ac.uk/˜jds31/research/ssavssi.pdf.

[26] J. Singer. Static single information improves type-based decompilation, 2003. http://www.cl.cam.ac.uk/˜jds31/research/ssidecomp.pdf.

[27] M. D. Smith. Extending SUIF for machine-dependent optimizations. In *Proceedings of the First SUIF Compiler Workshop*, pages 14–25, 1996. http://www.eecs.harvard.edu/machsuif/.

[28] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 108–120, 2000.

[29] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr 1991.

[30] D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: representation without taxation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–310, 1994.

```
  1: Translate(node) =
  2:   let C be the children of node in the dominator tree
  3:   let p_1, ..., p_n be the nodes of C that have more than one predecessor
  4:   for i ← 1 to n
  5:     let a_1, ..., a_k be the targets of φ-functions in p_i (possibly k = 0)
  6:     let S_i = Translate(p_i)
  7:     let F_i = "function f_{p_i}(a_1, ..., a_k) = S_i"
+ 8:   let s_1, ..., s_m be the nodes of C that are the target of a conditional branch
+ 9:   for i ← 1 to m
+ 10:    let q_i be the (unique) predecessor of s_i
+ 11:    let a_1, ..., a_k be the targets (associated with s_i) of σ-functions in q_i
+ 12:    let T_i = Translate(s_i)
+ 13:    let G_i = "function f_{s_i}(a_1, ..., a_k) = T_i"
! 14:  let F = F_1 F_2 ... F_n G_1 G_2 ... G_m
  15:  return Statements(node, 1, F)

  16: Statements(node, j, F) =
  17:  if there are < j statements in node
  18:  then let s be the successor of node
  19:    if s has only one predecessor
  20:    then return Statements(s, 1, F)
  21:    else s has m predecessors
  22:      suppose node is the ith predecessor of s
  23:      suppose the φ-functions in s are
             a_1 ← φ(a_1 1, ..., a_1 m), ...
             a_k ← φ(a_k 1, ..., a_k m)
  24:      return "let F in f_s(a_1 i, ..., a_k i)"
  25:  else if the jth statement of node is a φ-function
  26:    then return Statements(node, j + 1, F)
+ 27:  else if the jth statement of node is a σ-function
+ 28:    then return Statements(node, j + 1, F)
  29:  else if the jth statement of node is "return a"
  30:    then return "let F in return a"
  31:  else if the jth statement of node is a ← b ⊕ c
  32:    then let S = Statements(node, j + 1, F)
  33:      return "let a = b ⊕ c in S"
  34:  else if the jth statement of node is a ← b
  35:    then let S = Statements(node, j + 1, F)
  36:      return "let a = b in S"
  37:  else if the jth statement of node is "if a < b then goto s_1 else goto s_2"
  38:    then since this is edge-split SSI form
  39:    assume s_1 and s_2 each has only one predecessor
! 40:    let a_1, ..., a_k be
!          the source operands of σ-functions in node (possibly k = 0)
! 41:    return "let F in if a < b then f_{s_1}(a_1, ..., a_k) else f_{s_2}(a_1, ..., a_k)"
```

FIGURE 5.6 Algorithm that transforms SSI to functional intermediate representation

FIGURE 5.7 Dynamic data flow graph for three factorial(5) functions

Chapter 6

Implementing Mobile Haskell

André Rauber Du Bois[1], Phil Trinder[1], Hans-Wolfgang Loidl[2]

Abstract Mobile computation enables computations to move between a dynamic set of locations and is becoming an increasingly important paradigm. Mobile Haskell (*m*Haskell) is an extension of Haskell that supports mobile computation in open distributed systems i.e. dynamically changing systems where multiple executing programs can interact using a predefined protocol. This paper outlines the *m*Haskell primitives, discusses the design and pragmatics of their implementation and includes preliminary performance comparisons with Jocaml. The implementation addresses several challenges, including serialisation of programs in a lazy language with sharing and using a combination of bytecode and machine code to manage the *common software base*, i.e. to determine what to communicate between locations.

6.1 INTRODUCTION

Mobile Haskell [6] (*m*Haskell) is an extension of the purely functional Haskell language designed to facilitate the construction of distributed mobile software. As depicted in Fig. 6.1, *m*Haskell extends Concurrent Haskell [21], an extension supporting concurrent programming, with higher order communication channels called *Mobile Channels* (MChannels), that allow the communication of arbitrary Haskell values including functions, IO actions and channels.

The main features of the *m*Haskell implementation are:

- *m*Haskell *supports the construction of open systems*, enabling programs to connect and communicate with other programs and to discover new resources in the network. The abstractions we use to provide this basic functionality are

[1]School of Mathematical and Computer Science, Heriot-Watt University, Edinburgh EH14 4AS, Scotland, Email:`{dubois,trinder}@macs.hw.ac.uk`

[2]Ludwig-Maximilians-Universität München, Institut für Informatik, D 80538 München, Germany, Email: `hwloidl@informatik.uni-muenchen.de`

MChannels and remote evaluation and both have fast implementations in the RTS (runtime system) using C and TCP/IP sockets.

- *m*Haskell *is portable*. It is implemented as an extension of the GHC (*Glasgow Haskell Compiler*) [10] compiler that has been ported to many different architectures and operating systems. Our extensions are implemented using standard C and TCP/IP sockets, maintaining a high degree of portability.

- *m*Haskell *is designed to run on heterogeneous networks*. Mobile languages designed to work on global distributed systems, such as the Internet, must be able to communicate code between machines of different architectures and operating systems. The usual approach for communicating computations on heterogeneous networks is by compiling programs into architecture-independent byte-code. GHC combines both an optimising compiler and an interactive environment called GHCi, which compiles user defined functions into byte-code, and this technology could be used by *m*Haskell for communicating computations on heterogeneous networks.

- *m*Haskell takes a *hybrid approach*, combining byte-code and machine code. GHCi is designed for fast compilation and linking. It generates machine independent byte-code that is linked to the fast native-code available for the basic primitives of the language. As the basic modules in GHC are compiled into machine code and are present in every standard installation of the compiler, the routines for communication have to send only the machine independent part of the program and link it to the local definitions of the machine dependent part when the code is received. This gives us the advantage of having much faster code than using only byte-code.

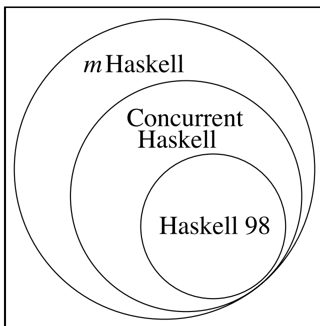

FIGURE 6.1 ***m*Haskell is an extension of Concurrent Haskell**

This paper is organised as follows: In the next section we present the MChannels communication primitives and the primitives for resource discovery and registration. In section 6.3 the implementation of *m*Haskell is described, first by giving a general overview of the platform and its challenges and then by describing

each of the design decisions. Finally, the low level issues of the implementation are discussed in section 6.4.

6.2 MOBILE HASKELL

6.2.1 Communication Primitives

Fig. 6.2 shows the MChannel primitives. Haskell with Ports [12] has similar primitives but restricts the type of values that can be communicated to basic values and data types, no functions or IO computations can be communicated.

```
data MChannel a      -- abstract
type HostName = String
type ChanName = String

newMChannel       :: IO (MChannel a)
writeMChannel     :: MChannel a -> a -> IO ()
readMChannel      :: MChannel a -> IO a
registerMChannel :: MChannel a -> ChanName -> IO ()
unregisterMChannel:: MChannel a -> IO()
lookupMChannel    :: HostName -> ChanName ->
                            IO (Maybe (MChannel a))
```

FIGURE 6.2 Mobile Channels

The `newMChannel` function is used to create a mobile channel and the functions `writeMChannel` and `readMChannel` are used to write/read data from/to a channel. MChannels are synchronous and have similar semantics to Concurrent Haskell channels: when a value is written to a channel the current thread blocks until the value is received in the remote host. In the same way when a `readMChannel` is performed in an empty MChannel it will block until a value is received on that MChannel. The functions `registerMChannel` and `unregisterMChannel` register/unregister channels in a name server. Once registered, a channel can be found by other programs using `lookupMChannel` which retrieves a mobile channel from the name server. A name server is always running on every machine of the system and a channel is always registered in the local name server with the `registerMChannel` function. MChannels are single-reader channels, meaning that only the program that created the MChannel can read values from it. Values are evaluated to normal form before being communicated.

Fig. 6.3 depicts a pair of simple programs using MChannels. First a program running on a machine called `ushas` registers a channel `mv` with the name `"myC"` in its local name server. When registered the channel can be seen by other ma-

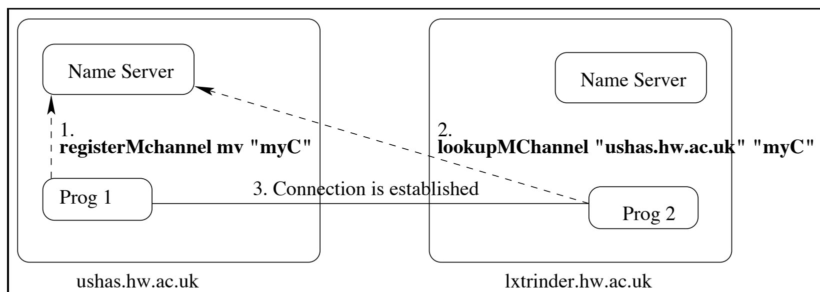

FIGURE 6.3 Example using MChannels

chines using the `lookupMChannel` primitive. After the lookup, the connection between the two machines is established and communication is performed with the functions `writeMChannel` and `readMChannel`.

6.2.2 Discovering Resources

One of the objectives of mobile programming is to better exploit the resources available in a network. Hence, if a program migrates from one node of the network to another, this program must be able to discover the resources available at the destination. By resource, we mean *anything* that the mobile computation would like to access in a remote host, from simple files to databases.

```
type ResName = String

registerRes :: a -> ResName -> IO ()
unregisterRes :: ResName -> IO ()
lookupRes :: ResName -> IO (Maybe a)
```

FIGURE 6.4 Primitives for resource discovery

Fig. 6.4 presents the three *m*Haskell primitives for resource discovery and registration. All machines running *m*Haskell programs must also run a registration service for resources. The `registerRes` function takes a name (`ResName`) and a resource (of type `a`) and registers this resource with the name given. The function `unregisterRes` unregisters a resource associated with a name and `lookupRes` takes a `ResName` and returns a resource registered with that name in the *local* registration service. To avoid a type clash, if the programmer wants to register resources with different types, she has to define an abstract data type that will hold the different values that can be registered.

A better way to treat type clashes would be to use dynamic types like Clean's *Dynamics* [22], but at the moment there is no complete implementation of it in any of the Haskell compilers.

6.2.3 Remote Thread Creation

*m*Haskell also provides a construct for remote thread creation:

```
rforkIO :: IO () -> HostName -> IO ()
```

It is similar to Concurrent Haskell's `forkIO` as it takes an IO action as an argument but instead of creating a local thread it sends the computation to be evaluated on the remote host `HostName`. The `rforkIO` function is implemented using MChannels as described in [6].

6.2.4 A Simple Example

Fig. 6.5 shows an *m*Haskell program that computes the load of a network. It visits a `listomachines` and executes the computation called `mobile` on all the machines of the list. First a channel `mch` is created and registered with the name `"mainmch"`. This channel is used by the remote locations to send the result of the computation back to the main machine. Then, the function `sendMobile` is mapped over the `listofmachines`. This computation looks for a specific channel called `clientmch` on the remote host and sends `mobile` to be executed remotely. The client receives the computation, executes it and sends the result back to the main program through the `mch` channel.

The program in figure 6.5, although simple, uses all the facilities provided by *m*Haskell (i.e. remote MChannels, registration of resources and mobile computation), and is used in the measurements given in Sec. 6.5.

6.3 IMPLEMENTATION DESIGN

6.3.1 Introduction

Mobile systems must abstract over the heterogeneity of large scale distributed systems, allowing machines with different architectures and different operating systems to communicate. This abstraction is usually achieved by compiling programs into architecture-independent byte-code. As a platform to build our system, we have chosen the *Glasgow Haskell Compiler* (GHC) [10], a state-of-the-art implementation of Haskell. The main reason for choosing GHC is that it supports the execution of byte-code combined with machine code. GHC is both an optimising compiler and an interactive environment called GHCi. GHCi is designed for fast compilation and linking. It generates machine independent byte-code that is linked to the fast native-code available for the basic primitives of the language. Both GHC and GHCi share the same runtime-system, based on the Spineless Tagless G-machine (STG)-machine [20], that is a graph reduction machine.

This and the next section explain the implementation of *m*Haskell using the GHC compiler. In this section, we discuss some design options at the language level and their influence on an implementation. In the next section we discuss the low level issues of the implementation.

```
main = do
  mch <- newMChannel
  registerMChannel mch "mainmch"
  list <- mapM (sendMobile mobile mch) listofmachines
  let v = sum list
  print ("Total Load of the network: " ++ (show v))
  where
   mobile = do
     res <- lookupRes "getLoad"
     case res of
     Just getLoad -> do
                     load <- getLoad
                     return load
     Nothing      -> return 0
   listofmachines = (...)

sendMobile:: IO() -> MChannel Int -> HostName -> IO Int
sendMobile comp mch host = do
  mc <- lookupMChannel host "clientmch"
  case mc of
    Just nmc -> writeMChannel nmc comp
  result <- readMChannel mch
  return result
```

FIGURE 6.5 Program that computes the load of a network

6.3.2 Evaluating Expressions before Communication

When a value is sent through a channel, it is evaluated before communication occurs. The reason for this design decision is that lazy evaluation makes it difficult to reason about what is being communicated. Consider the following example:

```
let
 (a,b,c) = f x
in if a then
    writeMChannel ch b
```

Suppose that the first element (`a`) of the tuple returned by `f x` is a Boolean, the second (`b`) an integer, and the third (`c`) is a large data structure. Based on the value of `a`, the program selects to send the integer `b` (and only `b`) to a remote host. In the example, it seems that the only value being sent is the integer, but because of lazy evaluation that is not what happens. In the beginning of the evaluation, the expression is represented by a graph similar to the one in figure 6.6.

At the point where `writeMChannel` is performed, the value `b` is represented in the heap as the *selector* that gets the second value of a tuple applied

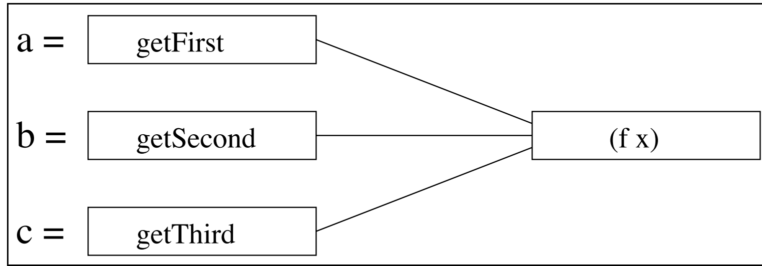

FIGURE 6.6 Graph for `let (a,b,c) = f x`

to the whole tuple. If `writeMChannel` does not evaluate its argument before communication, the whole value is communicated and this is not apparent in the Haskell code.

The evaluation of thunks (unevaluated expressions) affects only pure expressions or expressions that can be evaluated using `seq` (a Haskell function that evaluates its argument to *weak head normal form* (WHNF)). IO computations will not be executed during this evaluation step.

There are still ways of sending pure expressions to be evaluated on remote hosts. A tuple with a function and its arguments can be sent, and the function is applied to the values only on the remote end. Unevaluated expressions can also be communicated if wrapped in an IO value, as in the `apply` function:

```
apply :: (a->b) -> a -> IO b
apply f x = return (f x)
```

6.3.3 Sharing Properties

Many non-strict functional languages are implemented using graph reduction, where a program is represented as a graph and the evaluation of the program is performed by rewriting the graph. The graph ensures that shared expressions are evaluated at most once [19].

Maintaining sharing between nodes in a distributed system would result in a large number of extra-messages and call-backs to the machines involved in the computation (to request structures that were being evaluated somewhere else or to update these structures). In a typical mobile application, the client will receive some code from a channel and then the machine can be disconnected from the network while the computation is being executed (consider a handheld or a laptop). If we preserve sharing, it is difficult to tell when a machine can be disconnected, because even though the computation is not being executed anymore, the result might be needed by some other application that shares the same graph structure. The problem is already partially solved by making the primitives strict: expressions will be evaluated just once and only the result is communicated. In *m*Haskell, computations are *copied* between machines and no sharing is preserved.

6.3.4 MChannels

MChannels are single-reader channels, for two main reasons. First, it is difficult to decide where a message should be sent when we have more than one machine reading values from the same channel. The main question is where this channel is located. Channels with multiple readers need to maintain a distributed state, keeping track of all the machines that have references to the channel, and these references must be updated every time the channel is moved to another place.

A simple way to have multiple reader channels would be to keep the channel in one place, the place where it was created, and all other references to the channel read and write values into the channel by sending messages to this main location. The problem with this approach is that if the main location crashes all the other machines that have references to the channel cannot communicate anymore (Fig. 6.7).

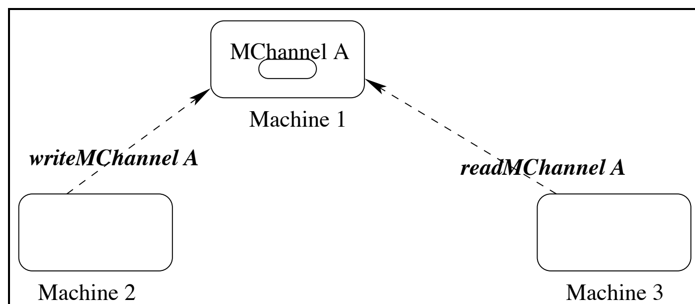

FIGURE 6.7 Machines 2 and 3 cannot communicate if Machine 1 crashes

The second reason is security: with multiple reader channels one process can *pretend* to be a server and steal messages. This is a classic problem also found in the untyped π-calculus [15].

6.4 THE IMPLEMENTATION

6.4.1 Packing Routines

The graph representing the computation being communicated is packed at the source and unpacked at the destination. The *m*Haskell pack and unpack routines are based on the GUM [26] system, but are extended to pack GHCi's Byte-Code Objects (BCOs).

Packing, or *serialising*, arbitrary graph structures is not a trivial task and care must be taken to preserve sharing and cycles. As in GPH [26], GDH [23] and Eden [1], packing is done breadth-first, closure by closure and when the closure is packed its address is recorded in a temporary table that is checked for each new closure to be packed to preserve sharing and cycles. We proceed packing until every reachable graph has been serialised.

The main heap object to be packed in our implementation of *m*Haskell is the BCO, that is GHC's internal representation for its architecture-independent byte-code. A BCO is composed of its `info_table` (which contains information about the closure's fields and also its entry code), a list of instructions, a list of pointers and a list of info tables. The BCO's info table is the same for every BCO so it does not need to be packed. Its list of instructions is just a list of bytes and is packed easily. The list of pointers contains a list of other closures that are used in the byte-code instructions, so all of them must also be packed. The list of info tables contains pointers to info tables of data structures that are constructed during the execution of the BCO's instructions. Those info tables are machine dependent hence are packed in a special way explained in section 6.4.2.

As the basic modules that come with GHC are compiled into machine code and are present in every standard installation of the compiler, the packing routines have to pack only the machine independent part of the program and link it to the local definition of the machine dependent part when the code is received and unpacked. This gives us the advantage of having much faster code than using only byte-code. Once packed, the BCO can be communicated in the way described in section 6.4.4. All machines running the mobile programs should have the same version of the GHC/GHCi system with an implementation of the primitives for mobility and also have the same binary libraries installed. Programs that communicate functions that are not in the standard libraries must be compiled into byte-code using GHCi.

Our packing mechanism gives us a simple way of controlling the amount of code communicated: since only functions that are compiled into byte code are packed, if the programmer knows that one module used in the computation is already in the remote host, this module must be compiled into machine code, so it will not be communicated.

Programs that will only receive byte-code do not need to have GHCi installed because the byte-code interpreter is part of GHC's RTS. In fact, if only functions from the standard libraries are used in the mobile programs, there is no need to have GHCi at all in both ends of the communication.

6.4.2 Communicating User Defined Types

Currently, user defined data types (ADTs) are always compiled into machine code in GHCi. There are two ways to overcome this problem. The first one would be to compile the types into a different type of closure that uses BCOs internally. This requires changing the compiler. The other solution is to ship the data type including the values in its info table. The entry code for these objects is very simple and has to be generated again in the destination.

In our current implementation, all data types used in the mobile programs must be defined in all the machines that are going to receive the code. Thus we only pack the name representing its info table in the linker and the content of its fields. When unpacking, we look for the local definition of the info table by searching for its name in the linker's tables. We consider an implementation of

one of the two solutions described above, as a tuning step in the development of the prototype implementation, aiming to reduce the common software base needed on all machines.

6.4.3 Evaluating Expressions

Evaluating expressions before communication is not as trivial as it seems. A simple way to evaluate thunks would be to use *evaluation strategies* [25], e.g.:

```
let list = [1..100]
in writeMChannel mch list
```

where in the definition of `writeMChannel` we use the `rnf` strategy to evaluate its argument to normal form.

But strategies will not work in all cases. Consider the following example:

```
f:: a -> b -> Int

let
  a = (...)
in writeMchannel ch (f a)
```

In this case it is not possible, inside of the definition of `writeMChannel`, to evaluate the expression `a` using strategies. One solution to this problem would be to implement a function `kids` with type:

```
kids:: HValue -> Array# HValue
```

That takes a value from the heap (the expression to be evaluated) and returns an array with all the thunks pointed to by this value. Using `kids` we can write a `deepSeq :: a -> ()` function that recursively applies `seq` to all the thunks pointed by its argument.

Another way to evaluate thunks is to do it inside the RTS using a primitive function that creates a new RTS thread to evaluate its argument to normal form by forcing the evaluation of all the expressions pointed by the argument.

*m*Haskell uses a hybrid approach: a thunk in the top level of the graph representing the computation is forced by a `seq` (as in Fig. 6.8). If there are other thunks in the graph, these thunks are evaluated by an extra thread in the RTS. Care must be taken to preserve the queue of closures yet to be packed if the new thread induces garbage collection. The solution to this problem is to make the packing queue visible to the Garbage Collector.

6.4.4 Implementation of MChannels

The basic structure to support MChannels is implemented in a similar way to Ports in Distributed Haskell [24].

Communication is implemented using the standard sockets library provided by the operating system, thus avoiding the need for any extra libraries like PVM

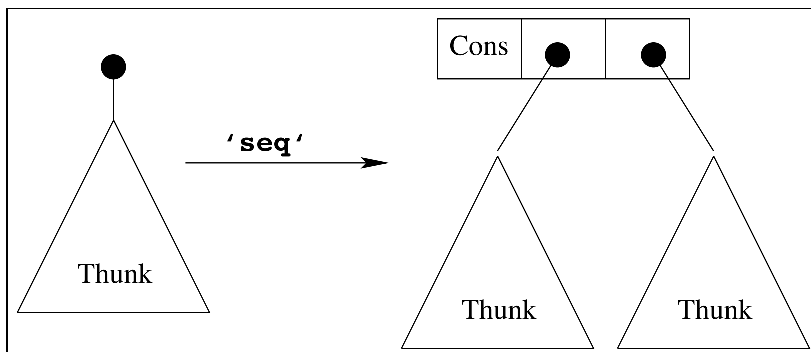

FIGURE 6.8 Evaluation of thunks using `seq`

or MPI. Haskell objects are serialised using the packing routines explained before and converted into an array of bytes that can be easily communicated through a socket.

Communication via sockets may use two different protocols: TCP and UDP. UDP is a fast connectionless protocol that does not handle message loss. TCP on the other hand is a connection-based protocol, making it easier to implement communication with the cost of a little extra overhead. We have chosen to implement the communication routines using TCP.

The channel data type is a simple Haskell data type that contains internally all the information that will be needed for communication, i.e. the name of the channel, the name of the host where it belongs and a concurrent Haskell channel (CHC) through which the communication between the program and the mobile runtime system occurs. When a new MChannel is created also a CHC is created to serve as a communication link between the program and the communication layer of the RTS. When a value is written into a MChannel, it is in fact written into its CHC. The RTS then reads this value from the CHC, serialises it and communicates it to the appropriate host based on the information present in the MChannel data type. When the RTS receives a value from a remote host this value is written into the CHC that represents the MChannel that should receive the message. A thread that reads a value from a MChannel is in fact reading a value from the internal CHC and will stay blocked in this CHC until a value is written by the RTS there.

To make ports visible to other machines in the network we use the `registerMChannel` and `lookupMChannel` primitives. These primitives communicate with an external naming service that keeps listening for requests on a well-known port. This service maintains a table with all the ports registered in the machine in which it is running. It also communicates with lookups launched by other hosts looking for channels. When a lookup is received, all the information about the channel is sent back to the client, so the client can communicate directly with the program that is waiting for requests on that channel.

TABLE 6.1 Comparative Jocaml and *m*Haskell Execution Times

Number of Machines visited	Jocaml (sec)	*m*Haskell (sec)
1	0.05s	0.47s
2	0.06s	0.93s
4	0.10s	1.85s
8	0.16s	3.70s
16	0.28s	7.42s

6.5 INITIAL EVALUATION

Table 6.1 shows a comparison between Jocaml [4] and *m*Haskell using the mobile program from section 6.2.4.

Jocaml [4] is an extension to Objective-Caml [17], a strict functional language with extensions for object-oriented programming, used to develop systems with mobile agents. Jocaml extends Objective-Caml with a small set of primitives taken from the Join-Calculus [8]. Jocaml programs communicate and synchronise through messages sent on channels, called *names* in the Join-Calculus terminology.

Although *m*Haskell presents good scalability when the number of machines is increased, it is still approximately 20 times slower than Jocaml. The main reason for that is the routine that recursively traverses the graph, forcing the evaluation of thunks before packing. Every time a computation is sent, the graph has to be traversed twice: once to force the evaluation and once for packing. It is not an option to force the evaluation while packing because the evaluation of the graph might change what has been already packed. Because Jocaml is strict, the evaluation of expressions to be communicated occurs naturally. Moreover, Jocaml is built as an extension to the Objective Caml compiler [17], a compiler with primitives for serialisation.

*m*Haskell is still in its early stages and a lot of optimisation could be applied. For example, in the program used in the experiments, the same function is sent to different hosts and is repacked every time it is communicated. Such packed computations could be stored for reuse.

6.6 RELATED WORK

There are numerous parallel and distributed Haskell extensions [27], and only those closely related to *m*Haskell are discussed here.

GPH and Eden are simple and powerful extensions to the Haskell language for parallel computing. They both allow remote execution of computation, but the placement of threads is implicit. The programmer uses the `par` combinator in GPH, or process abstractions in Eden, but where and when the data will be shipped is decided by the implementation of the language.

GDH is closer to the language presented here. Communication can be implemented using MVars and remote execution of computations is provided with the `revalIO` (remote evaluation) primitive. The problem in using GDH for mobile computation is that it is implemented to run on closed systems. After a GDH program starts running, no other PE (*processing element*) can join the computation. Moreover the GDH implementation relies on a *virtual shared heap* that is shared by all the machines running the computation. The algorithms used to implement this kind of structure will not scale well for very large distributed systems like the Internet [6].

Haskell with ports is a very interesting model to implement distributed programs in Haskell because it was designed to work on open systems. The only drawback is that the current implementation of the language restricts the values that can be sent through a port to the basic types and types that can instantiate the `Show` class. Furthermore, the types of the messages that can be received with `readPort` must be an instance of the `Read` class. The reason for these restrictions is that the values of the messages are converted to strings in order to be sent over the network [12].

There are other extensions to functional languages that allow the communication of higher-order values. Kali-Scheme [2] and Erlang [7] are examples of strict weakly typed languages that allow the communication of functions. Haskell is a statically typed language hence the communication between nodes can be described as a data type and many mistakes can be caught during the compilation of programs. Other strict typed languages such as Nomadic Pict [29], Facile [14] and Jocaml [4] implement the communication primitives as side effects while we integrate them to the IO monad, preserving referential transparency.

Curry [11] is a functional logic language that provides communication based on Ports in a similar way to the extension presented in this paper. Goffin [3] is a Haskell extension for concurrent constraint programming using ports but there is no distributed implementation of the language available yet. Another language that is closely related to our system is Famke [28]. Famke is an implementation of threads for the lazy functional language Clean [16] (using monads and continuations), together with an extension for distributed communication using ports. Famke has only a restricted form of concurrency, providing interleaved execution of atomic actions using a continuations monad.

6.7 CONCLUSIONS AND FUTURE WORK

We have presented the implementation of *m*Haskell, an extension of Haskell for mobile computation in open distributed systems. Unlike related systems, *m*Haskell can communicate arbitrary values, including functions and MChannels, between processors. This enables the use of powerful abstraction mechanisms provided by functional languages. Although the current implementation of *m*Haskell is still a prototype, it demonstrates the use of such abstraction mechanisms.

There are a number of issues that could be investigated in the future:

- It may be possible to extend the compiler with a *mobility analyses* (maybe based on a non-determinism analyses [18]) that would decide the parts of the program that should be compiled into byte-code and the parts that could be compiled into machine code, based on the occurrences of `writeMChannel`, as in [13].
- The implementation could be optimised, e.g. maintain a cache of functions already communicated to avoid repeated communication.
- Some languages that support mobility of code also support the migration of running computations (usually referred as *strong mobility* [9]). We could also extend Haskell with a primitive for transparent strong mobility that would be a primitive to explicitly migrate threads:

  ```
  moveTo :: HostName -> IO()
  ```

  The primitive `moveTo` receives as its argument a `HostName` to where the current thread should be moved.

  Strong mobility could be implemented in two ways: RTS level and Code Transformation.

  – RTS level: The state of the current thread (its stack) is packed and sent to be evaluated on a remote host. This work would extend our previous work on thread migration for the parallel functional language GPH [5].

  – Code Transformation: During compilation a program using `moveTo` is transformed into a simpler program that uses only weak mobility. One way to do that is to lift the IO monad into a continuation monad and then every call to `moveTo` is translated into a remote evaluation of the continuation of the current thread.

ACKNOWLEDGEMENTS

The authors would like to thank Simon Marlow and Simon Peyton Jones for their helpful comments on the implementation design for *m*Haskell. Bernard Pope also gave important suggestions about the evaluation of thunks. This work has been partially supported by an ORS and James Watt Scholarship.

REFERENCES

[1] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden Coordination Model for Distributed Memory Systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS)*, volume 1123. IEEE Press, 1997.

[2] H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(5):704–739, 1995.

[3] M. M. T. Chakravarty, Y. Guo, and M. Kohler. Distributed Haskell: Goffi n on the Internet. In *Fuji International Symposium on Functional and Logic Programming*, pages 80–97, 1998.

[4] S. Conchon and F. L. Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[5] A. R. Du Bois, H.-W. Loidl, and P. Trinder. Thread migration in a parallel graph reducer. In *IFL*, LNCS, Volume 2670. Springer-Verlag, 2002.

[6] A. R. Du Bois, P. Trinder, and H.-W. Loidl. Towards a Mobile Haskell. In *Proc. of the 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)*, pages 113–116, Valencia (Spain), 2003.

[7] Erlang. http://www.erlang.org/, WWW page, 2004.

[8] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Conference on Lisp and Functional Programming (LFP'84)*, Austin, Texas, 1996.

[9] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *Transactions on Software Engineering*, 24(5):342–361, May 1998.

[10] The Glasgow Haskell Compiler. http://www.haskell.org/ghc/, WWW page, 2004.

[11] M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, LNCS, Volume 1702, pages 376–395. Springer-Verlag, 1999.

[12] F. Huch and U. Norbisrath. Distributed programming in Haskell with ports. In *IFL*, LNCS, Volume 2011. Springer-Verlag, 2000.

[13] Z. D. Kirli. *Mobile Computation with Functions*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 2001.

[14] F. C. Knabe. *Language Support for Mobile Agents*. PhD thesis, School of Computer Science, Carnegie mellon University, 1995.

[15] R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, May 1999.

[16] E. Nocker, S. Smetsers, M. van Eekelen, and R. Plasmeijer. Concurrent Clean. In L. Aarts and Rem, editors, *Proc. of Parallel Architectures and Languages Europe (PARLE '91)*, pages 202–219. Springer-Verlag, 1991.

[17] OCaml. http://www.ocaml.org/, WWW page, June 2002.

[18] R. Peña and C. Segura. A polynomial cost non-determinism analysis. In *IFL*, LNCS, Volume 2312, pages 121–137. Springer-Verlag, 2002.

[19] S. Peyton Jones. *Implementation of Functional Programming Languages. A Tutorial*. Prentice Hall, 1992.

[20] S. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, 1992.

[21] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[22] M. Pil. Dynamic types and type dependent functions. In *Implementation of Functional Languages*, pages 169–185, 1998.

[23] R. Pointon, P. Trinder, and H.-W. Loidl. The design and implementation of Glasgow Distributed Haskell. In *IFL*, LNCS, Volume 2011. Springer-Verlag, 2000.

[24] V. Stolz and F. Huch. Implementation of Port-based Distributed Haskell. In *Draft. Proc. of IFL*, 2001.

[25] P. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998.

[26] P. Trinder, K. Hammond, J. S. Mattson Jr., A. S. Partridge, and S. L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadephia, USA, May 1996.

[27] P. Trinder, H.-W. Loidl, and R. Pointon. Parallel and distributed haskells. *Journal of Functional Programming*, 12(4/5):469–510, 2002.

[28] A. van Weelden and R. Plasmeijer. Towards a strongly typed functional operating system. In *IFL 2002*, 2002.

[29] P. T. Wojciechowski. *Nomadic Pict: Language and Infrastructure Design for Mobile Computation*. PhD thesis, Wolfson College, University of Cambridge, 2000.

Chapter 7

Testing Scheme Programming Assignments Automatically

Manfred Widera[1]

Abstract In distance learning the lack of direct communication between teachers and learners makes it difficult to provide direct assistance to students while they are solving their homework tasks. We address this problem particularly for programming tasks and describe a system for automatically analyzing students' homework tasks, and providing understandable feedback. Our approach is adapted to the special situation in distance learning and is integrated into the virtual university approach at the University of Hagen. It consists of a general framework and instances for individual programming languages. For these instances, one example is presented for the programming language Scheme.

7.1 INTRODUCTION

Both learning a programming language and giving a course in computer programming can be tedious tasks. A full programming language is usually a complex subject, so concentrating on some basic aspects first is necessary. One nice thing, however, about learning to program is that the student may get quick rewards, namely by seeing his own program actually being executed by a machine and (hopefully) getting the desired effects upon its execution. However, even writing a simple program and running it is often not so simple for beginners: many different aspects e.g. of the runtime system have to be taken into account, compiler outputs are usually not very well suited for beginners, and user manuals unfortunately often aim at the more experienced user.

In distance learning and education, direct interaction between students and tutors is particularly difficult. While communication via phone, e-mail, or news-

[1]Praktische Informatik VIII - Wissensbasierte Systeme, Fachbereich Informatik, FernUniversität in Hagen, 58084 Hagen, Germany; Email: manfred.widera@fernuni-hagen.de

groups helps, there is still need for more direct help in problem-solving situations like programming. In this context, intelligent tutoring systems have been proposed to support learning situations as they occur in distance education. A related area is tool support for homework assignments. In this paper, we will present an approach to the automatic revision of homework assignments in programming language courses. In particular, we describe a framework for analyzing programming homework tasks called AT(x) (analyze-and-test for a language *x*) and show how exercises in Scheme [7] can be analyzed and tested automatically by an instance AT(S) of it. The goal of AT(x) is to provide detailed generated feedback for the student. For the moment, automatic assessment of assignments is not provided by the system and is also only of minor importance for further extensions, compared to refined assistance for the students: while automatic assessment is a goal of interest in every area of teaching, the automatic assistance to the student is a special aim of distance learning and this system.

The destination platform for our AT(x) system is WebAssign [2, 6] which was developed at the University of Hagen for distance learning and is accessible for every teacher. WebAssign is a general system for support, evaluation, and management of homework assignments. Experiences with WebAssign, involving thousands of students over the last few years, show that especially for programming language courses (up to now mostly Pascal), the students using the system scored significantly higher in programming exercises than those not using the system. WebAssign is now widely used by many different universities and institutions [12].

Whereas WebAssign provides a general framework, customized components for different types of exercises are needed. For such components AT(x) provides an abstract frame which analyzes programs written by a student and – via WebAssign – sends back comments. In this way, AT(x) supports the learning process of our students by interaction that otherwise would not be possible. Apart from the general design of AT(x) and the benefits of such a generalized approach, in this paper we especially focus on the AT(x) instance AT(S) analyzing Scheme programs as an example for the analysis process on functional programming languages.

While WebAssign is the most important platform for the use of AT(x) in the near future, the system has also been coupled to VILAB, a virtual electronic laboratory for applied computer science [9]. VILAB is a system that guides students through a number of (potentially larger) exercises and experiments. The interface between AT(x) and VILAB is also generic over the different programming languages covered by the AT(x)-instances.

The rest of the paper is organized as follows: Sec. 7.2 gives an overview of WebAssign, the AT(x) system, and their interaction. A sample session of AT(S) analyzing a Scheme program is given in Sec. 7.3. Sec. 7.4 describes the general structure of AT(x) which consists of several components. The general requirements on an analysis component and their realization in the analysis component for Scheme programs are described in Sec. 7.5. Sec. 7.6 briefly states the current implementation and use of the system. In Sec. 7.7 related work is discussed, and conclusions and some further work are described in Sec. 7.8.

7.2 WebAssign AND AT(x)

The AT(x) system described in this paper is specialized to the situation at the FernUniversität in Hagen. For presenting and solving homework assignments online, the WebAssign system is available for use by customized assignment systems. Since WebAssign as the target platform had some influence on several design decisions for AT(x), we offer a brief overview of WebAssign and the way AT(x) is seen from WebAssign's point of view.

WebAssign is a web-based system that provides support for assignments and assessment of homework tasks. As stated in [2], it provides support with web-based interfaces for all activities occurring in the assignment process, e.g. for the activities of the author of a task, a student solving it, and a corrector correcting and grading the submitted solution. In particular, it enables tasks with automatic test facilities and manual assessment, scoring and annotation. WebAssign is integrated in the Virtual University system of the FernUniversität Hagen [10].

From the students' point of view, WebAssign provides access to the tasks to be solved by them. A student can work out his solution and submit it to WebAssign. Here, two different submission modes are distinguished. In the so-called *pre-test mode*, the submission is only preliminary. In pre-test mode, automatic analyses or tests are carried out to give feedback to the student. The student can then modify and correct his solution, and he can use the pre-test mode again until he is satisfied with his solution. Eventually, he submits his solution in *final assessment mode* after which the assessment of the submitted solution is done, either manually or automatically, or by a combination of both.

Several standard tasks are achieved by WebAssign and need not be addressed by customized analysis components using it.

- WebAssign provides *only* authenticated access for students and teachers. This can be based on a common authentication database for the whole university or on a database locally administered by WebAssign.

- Persistence of results between sessions and after final submission. In pre-test mode WebAssign stores the last submission for every task and every student in a database and provides it as a starting point to the student in further sessions. For final assessment, the teacher can access the solutions in this database (together with automatically generated comments if available). Comments and assessments from a human corrector or an automatic tool are also stored in this database and are made available to the student via WebAssign.

While WebAssign has built-in components for automatic handling of easy-to-correct tasks like multiple-choice questions, this is not the case for more complex tasks like programming exercises. Here, specific correction modules are needed. The AT(x) system (and especially the AT(S) instance described here in more detail) aims at analyzing solutions to programming exercises and is a system that can be used as an automatic correction module for WebAssign. Its main purpose is to serve as an automatic test and analysis facility in pre-test mode. (As a side-

effect we can make the output of the system available for the corrector in order to simplify the detection of errors.)

Instances of the AT(x) framework have a task database that contains an entry for each task. When a student submits a solution, AT(x) gets an assignment number identifying the task to be solved and a submitted program written to solve the task via WebAssign's communication components. Further information identifying the submitting student is also available, but its use is not discussed here. Taking the above data as input, AT(x) analyzes the submitted program. Again via WebAssign, the results of its analysis are sent as feedback to the student (cf. Fig. 7.1). The division of WebAssign and AT(x) is not only a logical one. While WebAssign is meant to reside on a global university server, the AT(x) components run on local servers that provide full control to individual teachers.

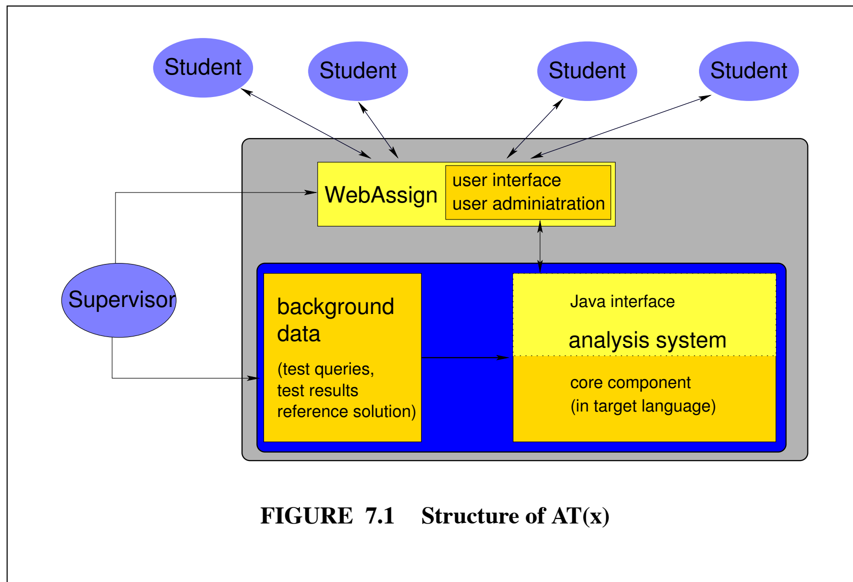

FIGURE 7.1 Structure of AT(x)

Owing to the learning situation in which we want to apply the analysis of Scheme programs, we did not make any restrictions with respect to the language constructs allowed in the students' solutions. AT(S) is able to handle the full standards of the Scheme programming language as it is implemented by MzScheme [4].

7.3 A SAMPLE SESSION

Before we go into the description of the individual components of the AT(x) system, we give an example of the execution of a homework task.

Solving a homework task includes the following subtasks: after logging into the WebAssign system the student chooses a task to solve in a web interface. The

task is presented as a web page containing forms for the solution. After filling in a solution (or correcting a previously supplied solution which is preserved between sessions), the student clicks a *submit* button. A few seconds later the system answers his submission with a new web page containing the analysis results. The submitted version replaces the previously preserved version of a solution.

Usually, several individual tasks are combined into an exercise. When the student is satisfied with all tasks in the exercise, he can close it, and the manual correction and assessment can start.

The following example is based on the AT(x) instance AT(S) for Scheme programs. The task is described as follows:

> Define a function `fac` that expects an integer n as input and returns the factorial of n if $n \geq 0$, and the atom `negative` otherwise.

Let us assume that the following program is submitted. After authentication has been performed by WebAssign, this is the only input the student has to pass to the system in order to solve the task.

```
(define (fac i)
  (if (= i 0) 1
      (+ i (fac (- i 1)))))
```

In this program the test for negative numbers is missing, and the first operator in the last line must be $*$ instead of $+$.
The system's output, identifying these two errors, is the following:

```
The following errors where detected in your program:
------------------------------------------------------
The following test was aborted to enforce termination:
(fac -1)
The function called when the abortion took place
was ''fac''.
A threshold of 10000 recursive calls was exceeded.
Please check whether your program contains an
infinite loop!
------------------------------------------------------
The following test was aborted to enforce termination:
(fac -42)
The function called when the abortion took place
was ''fac''.
A threshold of 10000 recursive calls was exceeded.
Please check whether your program contains an
infinite loop!
------------------------------------------------------
The following test generated a wrong result:
(fac 5)
The result generated was 16 instead of the
```

```
expected result 120.
------------------------------------------------------
The following test generated a wrong result:
(fac 6)
The result generated was 22 instead of the
expected result 720.
------------------------------------------------------
The following test generated a wrong result:
(fac 10)
The result generated was 56 instead of the
expected result 3628800.
------------------------------------------------------
```

One important aspect of the AT(S) system is the following: the system is designed to perform a large number of tests. In the generated report, however, it can filter some of the detected errors for presentation. Several different filters generating reports of different precision and length are available. The example above shows all detected errors (for a rather small test set) at once.

7.4 STRUCTURE OF THE AT(x) FRAMEWORK

The AT(x) framework combines different tools. Interfaces to different user groups (especially students and supervisors) have to be provided via WebAssign. The design decisions caused by this situation are described in this section.

7.4.1 Components of the AT(x) System

AT(x) is divided into two main components: the main work is done by the analysis component (lower part of the analysis system in Fig. 7.1). Especially in functional (and also in logic) programming, the used language is well suited for handling programs as data. The analysis component of AT(S) is therefore implemented in the target language Scheme.

A further component implemented in Java serves as an interface between this analysis component and WebAssign (upper part of the analysis system in Fig. 7.1). As shown in the figure, this interface completely performs the interaction between AT(x) and the WebAssign server. The reason for using such an interface component is its reusability and its easy implementation in Java. The WebAssign interface is based on Corba communication. A framework for WebAssign clients implementing an analysis component is given by an abstract Java class. Instead of implementing an appropriate Corba client independently for each of the AT(x) instances in the individual target languages, the presented approach contains a reusable interface component implemented in Java (that makes use of the existing abstract class) and a very simple interface to the analysis component.

The background data in Fig. 7.1 consists of text templates for error messages used by the interface and different static inputs to the core analysis component as described in Sec. 7.5.1.

7.4.2 Communication Interface of the Analysis Component

The individual analysis component is the main part of an AT(x) instance. It performs tests on the students' programs and generates appropriate error messages. The performed tests and the detectable error types of AT(S) are discussed in detail in Sec. 7.5. Here, we concentrate on the interface of this component.

The analysis component of each AT(x) instance expects to read an exercise identifier (used to access the corresponding information on the task to solve) and a student's program from the standard input stream. It returns its messages, each as a line of text, at the component's standard output stream. These lines of text contain an error number and some data fields containing additional error descriptions separated by a unique identifier. The number and the types of the additional data fields are fixed for each error number.

An example of such an error line is the following:

```
###4###(fac 5)###16###120###
```

Such a line consists of a fixed number of entries (four in this case) which are separated by ###. This delimiter also starts and ends the line. The first entry contains the error number (in this case 4 for a wrong result). The remaining entries depend on the error number. In this case, the second entry contains the test `(fac 5)` causing the error, the third one contains the wrong result 16, and the fourth one the expected result 120.

The presentation of the messages in a readable form is done by the Java interface component. An example of such a presentation is given in Sec. 7.3.

7.4.3 Function and Implementation of the Interface Component

WebAssign provides a communication interface based on Corba to the analysis components. In contrast, the analysis components of AT(x) use a simple interface with textual communication via the stdin and stdout streams of the analysis process, which avoids the need to re-implement a Corba client in the language used for the analysis component. We therefore use an interface process connecting an analysis component of AT(x) to WebAssign which performs the following tasks:

- Starting the analysis system and providing an exercise identifier and the student's program.
- Reading the error messages from the analysis component.
- Selecting some of the messages for presentation.
- Preparing the selected messages for presentation.

The interface component starts the analysis system (via the Java class *Runtime*) and writes the needed information into its standard input stream (which is available by the Java process via standard classes). Afterwards, it reads the message lines from the standard output stream of the analysis system, parses the individual messages and stores them into an internal representation.

During the implementation of the system it turned out that some language interpreters (especially SICStus Prolog used for the AT(P)-instance [1]) generate a number of messages at the stderr stream, e.g. when loading modules. These messages can block the analysis process when the stderr stream buffer is not cleared. Our Java interface component is therefore able to consume the data from the stderr stream of the controlled process without actually using them. With a minor change to the Java interface component the messages from stderr can, of course, be accessed. From our experience (using SICStus Prolog and MzScheme) it is, however, preferable to catch errors by custom error handlers inside the analysis components, providing appropriate messages via the standard interface of the analysis component. This keeps the interface between the two components uniform and avoids the need for parsing messages from the compiler that are usually not designed with automatic parsing in mind.

For presenting errors to the student, each error number is connected to a text template that gives a description of this kind of error. An error message is generated by instantiating the template of an error with the data fields provided by the analysis component together with the error number. The resulting text parts for the individual errors are concatenated and transferred to WebAssign as one piece of HTML text. An example of a message generated by the analysis component can be found in Subsec. 7.4.2. The sample session in Sec. 7.3 shows how this message is presented to the student.

When using this system in education it turns out that presenting all detected errors at once is not the best action in every case.

Example 7.1. Consider the example session described in Sec. 7.3. Having error messages for all detected errors available, a student could write the following program that only consists of a case distinction and easily outfoxes the system.

```
(define (fac n)
   (cond
     ((= n -1) 'negative)
     ((= n -42) 'negative)
     ((= n 5) 120)
     ((= n 6) 720)
     ((= n 10) 3628800)))
```

To avoid the kind of programs that are fine tuned to the set of tests performed by the analysis component, the interface component has the capability of selecting certain messages for output according to one of the following strategies:

- Only one error is presented. This is especially useful in beginners courses, since a beginner in programming should not get confused and demotivated by a large number of error messages. He can instead concentrate on one message and may receive further messages when restarting the analysis with the corrected program.
- For every type of error occurring in the list of errors only one example is selected for output. This strategy provides more information at once to ex-

perienced users. A better overview of the pathological program behaviour is given, because all different error types are described, each with one representative. This may result in fewer iterations of the cycle consisting of program correction and analysis. The strategy, however, still hides the full set of all test cases from the student and therefore prevents fine tuning a program according to the performed tests. Compared to returning just one message, this filter becomes more useful the more different errors can be distinguished.

- All detected errors are presented at once. This provides the complete overview over the program errors and is especially useful when the program correction is done offline. In order to prevent fine tuning of a program according to the performed tests, students should be aware that in final assessment mode additional tests not present in the pre-test mode will be applied.

Hiding some of the error messages and test cases from the student is, however, not a safe way to avoid fine tuned programs. Iterated testing with programs tuned towards all tests which are known so far eventually yields the whole set of test cases. Since the system is designed to support the students (and since e.g. a randomized test case generation needs special care to cover all special cases and is therefore quite complex), this weakness can be accepted for the purpose of AT(S).

7.4.4 Global Security Issues

Security is an issue that is common to all instances of AT(x). It should therefore be addressed by the framework rather than in every individual instance. Security includes the following topics:

- **Authentication:** access to WebAssign (apart from some introductory web pages) is only possible by authenticated users. User identificators are available with every submission. Since AT(x) is only accessible via WebAssign (using a Corba interface), and since WebAssign has proven its reliability during several years with thousands of students, further authentication is not necessary by AT(x).

- **Denial of service:** AT(x) is only accessed via WebAssign. The AT(x) system can therefore be protected by a firewall that can only be passed by the WebAssign server.

- **Malicious code from students:** without restricting the considered programming language, students' programs can access the machine running an AT(x) instance directly. Mechanisms preventing problems for the service include:
  - The analysis component can rule out malicious code. Here it is problematic to detect every malicious program without rejecting correct programs.
  - Several UNIX mechanisms can be employed to provide some relative form of security. It is possible to protect the machine and AT(x) itself, but a malicious program might still interfere with an analysis of another student's

program. This approach is implemented at the moment in AT(x) and was sufficient so far for programming exercises that do not need access to hardware.

– For system programming or other areas with extended need for security, a sandbox approach is necessary. In such an approach the interface component could start each instance of the analysis component in a new sandbox simulating the machines behaviour. Adapting or implementing such a sandbox is an area of future work in our implementation.

7.5 THE CORE ANALYSIS COMPONENT

The heart of the AT(x) system is given by the individual analysis components for the different programming languages. In this section we give an overview of the general requirements for these analysis components and describe a component for analyzing programs in Scheme instantiating AT(x) to AT(S) in more detail.

7.5.1 Requirements on the Analysis Components

The intended use in testing homework assignments rather than arbitrary programs implies some important requirements and properties of the analysis component discussed here: it can rely on the availability of a detailed specification of the homework tasks, it must be robust against non-terminating input programs and runtime errors, and it must generate reliable output understandable for beginners.

Though the requirements formulated here carry over to an extension towards automatic *assessment* (comparable to e.g. [5]) we especially focus on the goal of quick, reliable and understandable feedback given to the students.

The description for each homework task consists of the following parts:

- A textual description of the task. (This is not directly needed for analyzing students' programs. For the teacher it is, however, convenient in preparing the tasks to have the task description available together with the other data items described here.)
- A set of test cases for the task.
- A reference solution. (This is a program which is assumed to be a correct solution to the homework task and which can be used to judge the correctness of the students' solutions.)
- Specifications of program properties and of the generated solutions. (This is not a necessary part of the input. In our implementation we use abstract specifications mainly for Prolog programs (cf. [1]). They are, however, also available for AT(S).)

This part of input is called the *static input* to the analysis component because it usually remains unchanged between the individual test sessions. Each call to the analysis system contains an additional *dynamic input* which consists of a unique

identifier for the homework task (used to access the appropriate set of static input) and a program to be tested.

We now discuss the requirements on the behaviour of the analysis system in more detail. Concretizing the requirement of reliable output, we want our analysis component to return an error only if such an error really exists. Where this is not possible (especially when non-termination is suspected), the restricted confidence should clearly be communicated to the student, e.g. by marking the returned message as a *warning* instead of an *error*. For warnings the system should describe an additional task to be performed by the student in order to discriminate errors from false messages. Especially in checking generated results for correctness, special care has to be taken that all correct alternative solutions are considered correct.

Runtime errors of every kind must be caught without affecting the whole system. If executing the student's program causes a runtime error, this should not corrupt the behaviour of the other components. Towards this end, our AT(S) implementation exploits the hooks of user-defined error handlers provided by MzScheme [4]. An occurring runtime error is reported to the student, and no further testing is done, because the system's state is no longer reliable.

For ensuring termination of the testing process, infinite loops in the tested program must also be detected and aborted. As the question whether an arbitrary program terminates is undecidable in general, we chose an approximation that is easy to implement and guarantees every infinite loop can be detected: a threshold for the maximal number of function calls (counted independently for each function) is introduced and the program execution is aborted whenever this threshold is exceeded.[1] As homework assignments are usually small tasks, it is possible to estimate the maximal number of needed function calls and to choose the threshold sufficiently. The report to the student must, however, clearly state the restricted confidence on the detected non-termination.

Counting the number of function calls is only possible when executing the program to be tested in a supervised manner. The different approaches for supervising recursion include the implementation of an own interpreter for the target language; and the instrumentation of each function definition during a preprocessing step such that it calls a counter function at the beginning of every execution of the function. The second approach was chosen for AT(S) and is described in more detail in the following subsection.

7.5.2 Analysis of Scheme Programs

The aim of the AT(S) analysis component is the evaluation of tests in a given student's program and to check the correctness of the results. A result is considered correct if comparing it with the result of the reference solution does not indicate an error.

A problem inherent to functional programs is the potentially complex structure

[1] In the context of the Scheme programs considered here, every iteration is implemented by recursion and therefore supervising the number of function calls suffices. In the presence of further looping constructs, a refined termination control is necessary.

of the results. Not only can several results to a question be composed into a structure, but it is furthermore possible to generate functions (and thereby e.g. infinite output structures) as results.

Example 7.2. Consider the following homework task:

> Implement a function `words` that expects a positive integer n and returns a list of all words over the alphabet $\Sigma = \{0,1\}$ with length l, $1 \leq l \leq n$.

For the test expression `(words 3)` there are (among others) the valid solutions

```
(0 1 00 01 10 11 000 001 010 011 100 101 110 111)
(1 0 11 10 01 00 111 110 101 100 011 010 001 000)
(111 110 101 100 011 010 001 000 11 10 01 00 1 0)
```

which only differ in the order of the words. Since no order has been specified in the task description, all these results must be considered correct.

For comparing such structures, a simple equality check is not appropriate. Instead, we provide an interface for the teacher to implement an equality function that is adapted to the expected output structures and that returns true if the properties of the two compared structures are similar enough for assuming correctness in the context of pre-testing. Using such an approximation of the full equality is safe since in the usual final assessment the submission is corrected and graded by a human tutor. In order not to confuse the student it is, however, critical not to report correct results as erroneous, merely because they differ from the expected result.

Example 7.3. For the task in Example 7.2 the equality check could be

```
(define (check l1 l2)
  (equal? (sort l1) (sort l2)))
```

with an appropriate sort function `sort`.

A more complex test can e.g. consist of comparing functions from numbers to numbers. Such a test can return true after comparing the results of both functions for n (for some appropriate number n) well-chosen test inputs for equality. If an assignment is expected to return more complex functions, it is even possible to consider the returned function as new homework and to call the analysis component recursively, provided that the specimen program is given as a result for a new task.

Termination analysis of Scheme programs is done by applying a program transformation to the student program. We have implemented a function that counts the number of function calls for different lambda expressions independently and that aborts the evaluation via an exception if the number of calls exceeds a threshold for one of the lambda expressions. To perform the counting, each lambda expression of the form

```
(lambda (args) body)
```

is transformed into

```
(lambda (args) (let ((tester::tmp tc)) body))
```

where `tc` is an expression sending a message to the count function containing a unique identifier of the lambda expression and `tester::tmp` is just a dummy variable whose value is not used.

After performing the transformation on the student's program, the individual tests are evaluated in the transformed program and in the reference solution. The results from both programs are compared, and messages are generated when errors are detected. Runtime errors generated by the student's program are caught, and an explaining error message is sent to the interface component of AT(S).

In detail, the analysis component of AT(S) is able to distinguish several error messages, which can stem from failed equality checks, the termination control and the runtime system. These include wrong results generated by the student's program, aborted executions due to suspected infinite loops, syntax errors, undefined identifiers, and several other kinds of runtime errors detected by the system. A generic error code can be used by the system to give detailed descriptions on failed tests for certain program properties, e.g. factorial can be checked always to return a non-negative integer.

For each of these errors the interface component of AT(S) contains a text template that is instantiated with the details of the error, and is then presented to the student. When implementing a new instance of AT(x) an appropriate set of codes needs to be defined, and text templates for these codes have to be provided to the interface component by instantiating an abstract Java class.

7.6 IMPLEMENTATION AND EXPERIENCES

The AT(x) framework with its instance AT(S) (and a further instance for Prolog) is fully implemented and operational. The analysis component runs under the Solaris 7 operating system and, via its Java interface component, serves as a client for WebAssign.

Owing to the modular design of our system, the implementation of new analysis components can concentrate on the analysis tasks. The implementation of the analysis component of AT(S) took approximately three person months. For the adaption of the starting procedure and the specific error codes inside the interface component an additional two weeks were necessary.

At the moment the system with the instances AT(P) and AT(S) for Prolog and Scheme goes through its first application in a programming course. It is available only for selected homework tasks. Although using the system means sending in homeworks in two different ways (WebAssign for the selected available tasks, plain paper sent in by mail for the remaining tasks) two thirds of the active students used the system. Feedback from the students was positive in general, mentioning both a better motivation to solve the tasks and better insight in the new programming paradigm.

7.7 RELATED WORK

In the context of teaching Scheme, the most popular system is DrScheme [11]. The system contains several tools for easily writing and debugging Scheme programs by students. For testing a program, test suites can be generated. Our AT(S) system differs from that approach primarily in providing a test suite that is hidden from the student and that is generated without a certain student's program in mind, but following the approach called *specification based testing* in testing literature (cf. e.g. [13]).

A system very similar to our approach is presented in [5] for Ceilidh. While our approach is focused on quick and understandable feedback to the students, Ceilidh is used for automatic assessment of homework assignments. Since the WebAssign system offers automatic assessment, it might be possible to extend the scope of our system in this direction. Because of the undecidability of program equivalence and program correctness, however, we decided to run some tests with hand correction of assignments first, using the corresponding pre-correction outputs to simplify the manual final correction.

Other testing approaches to functional programming (e.g. QuickCheck [3]) do not focus on testing programming assignments and are therefore not designed to use a reference solution for judging the correctness of computation results. The approach of abstractly describing properties of the intended results can be found in our approach as well. The randomized generation of test cases used in QuickCheck is a possible extension of our system. We must, however, make sure that tests for special cases are contained in every test set.

A further topic related to our approach is the area of intelligent tutoring systems (ITS) (see e.g. an overview in [8]). Our approach does not aim at the goals of an ITS, but is just a testing tool to be integrated in the distance learning context of the FernUniversität in Hagen. Even when thinking of an "intelligent" testing tool, finding and understanding the errors in the student solutions is a first necessary step, so that our tool can be of use in constructing an ITS in future.

An automatic tool for testing programming assignments in WebAssign already exists for the programming language Pascal [12]. In contrast to our approach here, several different programs have to be called in sequence, namely a compiler for Pascal programs and the result of the compilation process. The same holds for possible analysis tools aiming at other compiled programming languages like e.g. C and Java. To keep a uniform interface, it is advisable to write an analysis component that compiles a program, calls it for several inputs, and analyzes the results. This component can then be coupled to our interface component instead of rewriting the interface for every compiled language. For instantiating AT(x) to another functional programming language it is, however, advisable to use the read-evaluate-print-loop of the language, and to implement the analysis component completely in the target language.

Putting the differences together, the AT(x) approach is novel in providing a framework that is highly generic over both the chosen programming language (with a focus on high-level languages providing a REP-loop) and the communi-

cation platform (up to now mostly WebAssign, but also VILAB). It is completely focused on aiding the student in solving programming tasks in a distance learning framework.

7.8 CONCLUSIONS AND FURTHER WORK

We addressed the situation of students in programming lessons during distance learning studies. The problem here is the usually missing or insufficient direct communication between learners and teachers and between learners. This makes it more difficult to get around problems during self-tests and homework assignments.

In this paper we have presented the AT(x) approach, which is capable of automatically analyzing programs with respect to given tests and a reference solution. In the framework of small homework assignments with precisely describable tasks, the AT(x) instances are able to find many of the errors usually made by students and to communicate them in a manner understandable for beginners in programming (in contrast to the error messages of most compilers.)

The AT(x) framework is designed to be used in combination with WebAssign, which is available at the FernUniversität Hagen, and provides a general framework for all activities occurring in the assignment process. This causes AT(x) to be constructed from two main components, an analysis component (often written in the target language) and a uniform interface component written in Java.

By implementing the necessary analysis components, instances of AT(x) for different programming languages are generated. This was presented for the instance AT(S), which performs the analysis task for Scheme programs. This analysis component is robust against programs causing infinite loops and runtime errors, and is able to generate appropriate messages in these cases. The general interface to WebAssign makes it easy to implement further instances of AT(x), for which the required main properties are also given in this paper.

During the next semesters, AT(S) will be applied in courses at the FernUniversität Hagen and its benefit for Scheme programming courses in distance learning will be evaluated.

Future work on AT(S) can address the following topics. While the current system aids the students in preparing their homework assignments, an automatic assessment stage comparable to [5] can reduce the effort required by the teacher to correct them. This, however, makes it necessary to understand errors not only in terms of the I/O-behaviour, but in terms of the source code. The precise assessment can be calculated as the similarity of the student's solution to a specimen program according to some appropriate distance function. Understanding errors in terms of the source code is also necessary in order to extend AT(S) towards an ITS. Furthermore, an ITS needs a model of the student's programming skills and possible misunderstandings, in order to find reasons for certain errors and to provide more specialized help. In all these extensions we believe that useful online assistance to the students should always be one of the most important aims (or even the most important aim) in distance learning.

REFERENCES

[1] C. Beierle, M. Kulaš, and M. Widera. Automatic analysis of programming assignments. In *Proceedings of the 1. Fachtagung "e-Learning" der Gesellschaft für Informatik (DeLFI 2003)*. Köllen Verlag, Bonn, 2003.

[2] J. Brunsmann, A. Homrighausen, H.-W. Six, and J. Voss. Assignments in a Virtual University – The WebAssign-System. In *Proc. 19th World Conference on Open Learning and Distance Education*, Vienna, Austria, June 1999.

[3] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 268–279, N.Y., Sept. 18–21 2000. ACM Press.

[4] M. Flatt. *PLT MzScheme: Language Manual*, Aug. 2003.

[5] S. Foubister, G. Michaelson, and N. Tomes. Automatic assessment of elementary standard ml programs using ceilidh. *Journal of Computer Assisted Learning*, 1996.

[6] A. Homrighausen and H.-W. Six. Online assignments with automatic testing and correction facilities (abstract). In *Proc. Online EDUCA*, Berlin, Germany, October 1997.

[7] R. Kelsey, W. Clinger, and J. R. (Editors). Revised^5 report on the algorithmic language scheme. *ACM SIGPLAN Notices*, 33(9):26–76, Sept. 1998.

[8] R. Lelouche. Intelligent tutoring systems from birth to now. *Künstliche Intelligenz*, 13(4):5–11, Nov. 1999.

[9] R. Lütticke, C. Gnörlich, and H. Helbig. Vilab - a virtual electronic laboratory for applied computer science. In *Proceedings of the Conference Networked Learning in a Global Environment*. ICSC Academic Press, Canada/The Netherlands, 2002.

[10] Homepage LVU, FernUniversität Hagen, `http://www.fernuni-hagen.de/LVU/`. 2003.

[11] *PLT DrScheme: Programming Environment Manual*, May 2003. version204.

[12] H. WebAssign. `http://www-pi3.fernuni-hagen.de/WebAssign/`. 2003.

[13] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.

Chapter 8

Testing Reactive Systems with GAST

Pieter Koopman and Rinus Plasmeijer [1]

Abstract G∀ST is a fully automatic test system. Given a logical property, stated as a function, it is able to generate appropriate test values, to execute tests with these values, and to evaluate the results of these tests. Many reactive systems, like automata and protocols, however, are specified by a model rather than in logic. There exist tools that are able to test software described by such a model-based specification, but these tools have limited capabilities to generate test data involving data types. Moreover, in these tools it is hard or even impossible to state properties of these values in logic. In this paper we introduce some extensions of G∀ST to combine the best of logic and model based testing. The integration of model based testing and testing based on logical properties in a single automated system is the contribution of this paper. The system consists only of a small library rather than a huge stand-alone system.

8.1 INTRODUCTION

Within the fully automatic test system G∀ST [15], properties over functions and data types are expressed in first order logic. These properties are written as functions in the functional programming language CLEAN [18]. Based on the types used in these functions, G∀ST automatically and systematically generates test values. It evaluates the property for these values and analyses the test results. This avoids the burden to design and evaluate a test suite by hand and makes it easy to repeat the test after changing the program (*regression tests*). This automatic and systematic generation of test data is a distinguishing feature of G∀ST that even allows proofs for finite types by exhaustive testing. In [15] we focused mainly on the concepts and implementation of G∀ST.

[1] Nijmegen Institute for Computer and Information Science, Nijmegen University, The Netherlands. Email: {pieter,rinus}@cs.kun.nl

It is possible to specify the behaviour of reactive systems, like the famous coffee-vending machines and protocols [5], in logic, as demonstrated by *Z* [20]. However, these reactive systems are usually specified by a model, instead of by a property in logic. Many formalisms are used in the literature to specify reactive systems. We use labelled transition systems (LTS), since they have shown to be very general and effective for testing [13, 10].

G∀ST was originally designed for logic-based testing, not for model-based tesing. In this paper we introduce some extensions that make G∀ST suitable for model based testing. We introduce a general format to specify labelled transition systems as a data structure in CLEAN. The specification of an LTS by a function is shown to be more concise and can handle an unbounded number of labels and states.

To test conformance effectively these specifications are used as a basis for test case generation. These test cases are much more effective for this purpose than the systematic generation of all possible inputs, which in its turn is more effective than random generation of inputs. For each deterministic and finite LTS it becomes possible to prove that the implementation behaves as specified, or to spot an error under the assumption that the implementation is an LTS that does not contain more states than the specification [25].

An advantage of extending G∀ST to enable testing of products specified by an LTS is that the original ability to test data types is preserved and can be combined with the new possibilities. The generation of data to test properties involving data types is a weak point of the existing automatic model-based test systems.

Unlike model checkers like SPIN [12], we assume that the given specification is correct. In practice, however, differences between the specification and the actual implementation appear also to be caused by incorrect specifications. So, testing also increases the quality and confidence in the specification.

8.2 OVERVIEW OF G∀ST

To make this paper self-contained we give an overview of G∀ST. It is an automatic test system embedded in the functional programming language CLEAN. The idea behind G∀ST is similar to the test system Quickcheck for Haskell [7, 8]. Distinguishing features of G∀ST are the systematic test data generation and the ability to prove properties. Quickcheck generates test data randomly.

Ordinary CLEAN functions are used to specify properties. As an example, we consider the *rotate 13* algorithm, a simple way to encrypt texts. It is used to hide text from casual reading and rotates the alphabet by half its length, i.e. 13 characters. Characters not in the alphabet are not effected. For example, the encryption of `The answer = 42` yields `Gur nafjre = 42` [1].

A nice property of this encryption method is that it is its own decryption: applying the algorithm twice yields the original character. In logic this is $\forall c \in$ $\texttt{Char}.rot13(rot13(c)) = c$. In G∀ST this is expressed as:

```
propRot13 :: Char -> Bool
propRot13 c = rot13 (rot13 c) == c
```

Notice that the arguments of the functions that specify the desired property are treated as universally quantified variables.

8.2.1 Testing and Results

Given an implementation of `rot13`, the property `propRot13` is tested by applying it for a number of characters and checking whether it yields **True** for all arguments. This is exactly what the function `test` does: generate arguments of the desired type in a systematic way, evaluate the specified property for these arguments, and investigate whether the test cases are successful. This test is initiated by executing `Start = test propRot13`. We use the following implementation of `rot13` in the tests:

```
rot13 :: Char -> Char
rot13 c | isUpper c = toChar ((toInt(c-'A')+13) rem 26) + 'A'
        | isLower c = toChar ((toInt(c-'a')+13) rem 26) + 'a'
                    = c
```

Testing this property yields: *`Proof: success for all arguments after 98 tests`*. Owing to the systematic generation of test data, G∀ST can, in this situation, detect that this property holds for all possible well–defined arguments. Hence the result qualifies as a proof rather than just a successful test result. For the type `Char` G∀ST only generates the printable characters; which explains why there are only 98 successful test performed. Below we show how this property is tested for all 256 possible characters, if that is desired.

8.2.2 Evaluating Test Results

The function `test` has type `p->[String]|Testable p`. Given a member of the class `Testable`, this function yields a list of strings containing the test report. There exist instances of the class `Testable` for `Bool` and functions of type `(a->b)|Testable b & TestArg a`. A type belongs to the class `TestArg` if G∀ST knows how to generate and show values of this type.

The basic rules for evaluating a series of test results are rather simple:

1. As soon as a single counterexample is encountered the property does not hold. The testing process terminates with an appropriate error message.

2. If no counterexamples are found and all possible test values are used, the property is proven. Such a proof is only possible for finite types and feasible for rather small types.

3. If no counterexamples are found within a certain upper bound of tests, the property passes the test successfully. We gained confidence in its correctness.

8.2.3 Logical Operators in G∀ST

As an additional property we might require that applying `rot13` to any character yields a different character:

```
propRot13b :: Char -> Bool
propRot13b c = rot13 c <> c
```

Testing this property yields the message: *Counterexample found after 5 tests: ';'*. As stated above, only alphabetic characters are changed. Other characters are unaffected by `rot13`. Hence `rot13 ';'` is equal to `';'` and this property does not hold for `';'`.

For a more precise formulation of this property we might require that applying `rot13` to a letter yields a different character:

```
propRot13c :: Char -> Property
propRot13c c = isAlpha c ==> rot13 c <> c
```

The operator `==>` mimics the implication operator, $\Rightarrow$, from logic. It has the usual semantics: if the left operand holds, the right-hand operand should be obeyed. For implementation reasons this function yields an element of type `Property` rather than a Boolean. Any Boolean result is transformed to such a `Property` by applying the function `prop`. Semantically the type `Property` is the union of Booleans and functions yielding a Boolean (which are just logical expressions containing a universal quantifier). Evaluating this property by G∀ST yields: *Proof: Success for all not rejected arguments, 52 tests, 46 rejections*.

If the left-hand argument of the operator `==>` yields `False`, the test-value is rejected instead of counted as success. This operator is used to select test values: if the test value is rejected, nothing is known about the property on the right-hand side. It would be misleading to count this as a successful test.

There are several ways for the tester to control the generation of test values. Using the infix operator `For` the property is tested for all values in the list on the right-hand side of the operator. The `For` operator is used to test `propRot13` for all 256 characters in the standard ASCII in:

```
Start = test (propRot13 For map toChar [0..255])
```

Here G∀ST reports *Passed after 100 tests*. In this situation it is easy to turn this result to a proof. We only have to increase the number of tests allowed.

```
Start = testn 500 (propRot13 For map toChar [0..255])
```

G∀ST reports *Proof: success for all arguments after 256 tests*.

8.2.4 Automatic Generation of Test Values

Test data generation for predefined types like `Char` is rather easy. G∀ST generates all possible elements of finite and relatively small types like `Bool` and `Char` as test value. For large types like `Int` and `Real` this is of course not feasible. G∀ST generates by default common border types (like -1, 0 and 1), followed by random values for these types.

The generation of test values for user-defined (recursive) types is interesting. Using CLEAN's generic programming facilities [11, 3], G∀ST generates instances of these types fully automatically. Test data are generated such that small instances come first and larger values afterwards. Owing to the use of systematic generation duplicates are also avoided in this situation. This implies that G∀ST is able to detect that all instances of a finite type are generated. If a property holds for all these values, it is proven correct.

8.3 SPECIFYING REACTIVE SYSTEMS IN G∀ST

A reactive system is an automaton that posseses an internal state and interacts with its environment. In this paper we restrict ourselves to software systems with a single input and output channel. For instance, a communication channel is modelled as a function of type `[Message] -> [Message]`.

For some simple reactive systems we can specify aspects of their behaviour in first order logic. For instance, a system consisting of an unreliable communication channel supervised by an alternating bit protocol is required to yield the same list of messages as is to be sent. In G∀ST this is:

```
propAltBit :: (Int->Bool) (Int->Bool) [Int] -> Bool
propAltBit sError rError input = input == abpSystem sError rError input
```

The function `abpSystem::(Int->Bool) (Int->Bool) [c] -> [c]` mimics the communication channel. The first two function arguments are used for the introduction of communication errors in the sending and receiving direction of the channel respectively. The last argument, the list `[c]`, is the input of the channel and the result is the output of the alternating bit protocol to the user.

The implementation of the alternating bit protocol used in the tests is:

```
:: Message c     = M c Bit | A Bit | Error
:: SenderState c = Send Bit | Wait Bit c

sender :: (SenderState c) [c] [Message c] -> [Message c]
sender (Send b) []     as = []
sender (Send b) [c:cs] as = [M c b: sender (Wait b c) cs as]
sender state=:(Wait b c) cs [a:as]
 = case a of
    A d | b==d = sender (Send (~b)) cs as
    _          = [M c b: sender state cs as]

receiver :: Bit [Message c] -> ([Message c],[c])
receiver rState [] = ([],[])
receiver b [m:ms]
 = case m of
    M c d|b==d = ([A   b :as],[c:cs]) where (as,cs) = receiver (~b) ms
    _          = ([A (~b):as],   cs ) where (as,cs) = receiver b ms

channel :: (Int->Bool) [Message c] -> [Message c]
channel error ms = [ if (error n) Error m \\ m <- ms & n <- [1..]]

abpSystem sError rError list = received
where   (acks,received) = receiver firstBit (channel sError messages)
        messages = sender (Send firstBit) list (channel rError acks)
        firstBit = O
```

This implementation passes any test of the property `propAltBit` in G∀ST.

Although this works fine, the properties that can be specified in this way are limited. For instance, it is troublesome to specify the behaviour of the sender of the alternating bit protocol in this formalism. Often, labelled transition systems are used to specify this kind of behaviour of systems.

8.3.1 Labelled Transition Systems

A very popular way to specify a reactive systems is by means of a labelled transition system (LTS). In this section we introduce labelled transition systems, show how they can be represented in CLEAN and show how they can be used as a basis for testing in our predicate based test system.

An LTS description is defined in terms of a set of states and labelled transitions between these states. To have a clear separation between input and output labels we deviate from the usual definition of an LTS by using different types. Moreover, we allow one input to generate a list of outputs. By introducing additional intermediate states, such an LTS can be transformed to a traditional LTS. Our representation reduces the number of transitions needed to specify a system and makes it easier to use an LTS as a basis for testing.

Given Q a non-empty countable set of states, I a non-empty countable set of input symbols, and O a non-empty countable set of output symbols, we have a transition relation $T \subseteq Q \times I \times \langle O \rangle \times Q$. Given some $q_0 \in Q$ a labelled transition system is give by the tuple (Q, I, O, T, q_0).

For the moment we restrict ourselves to deterministic systems: the output and new state are uniquely determined by the current state and the input. In fact we have a Mealy finite state machine [17]. That is, if $(q, i, o_1, q_1) \in T$ and $(q, i, o_2, q_2) \in T$ we have $q_1 = q_2 \wedge o_1 = o_2$. One often writes $(q_1, i, o, q_2) \in T$ as:

$$q_1 \stackrel{i/o}{\rightarrow} q_2$$

Where model checkers and other test systems often use a tailor-made specification language (like Promela used within SPIN [12] and TorX [22]) to describe the labelled transition systems that serves as specification, we prefer a specification in CLEAN. This has two advantages. First, we can use the full power of a functional programming language to write the specification or to write functions that generate the desired specification. Second, there is no need for an additional language.

Instead of explicit sets of states, Q, and labels, I and O, we employ the type system of CLEAN to enforce the correct use of states and labels. A straightforward realisation of an LTS consists of a record containing a list of transitions and an initial state.

```
:: Transition state input output :== (state,input,[output],state)
:: LTS state input output
    =   { trans     :: [Transition state input output]
        , initial   :: state
        }
```

The use of type-parameters for the sets of states and labels involved gives us

maximum flexibility. We can even use various different types of transition systems in the same program if desired.

Usually the LTS is a partial function, so we have to decide what to do when an input is received in a state that is not covered by the LTS. Like most model checkers we choose to ignore the input: the state does not change and the output is empty. This is known as *implicit completion* of the model.

8.3.2 Example: Conference Protocol

The conference protocol described here is a well-known case study in many model specifications and testers [24]. The conference protocol is used to describe the behaviour of a conference protocol entity (CPE). The conference protocol allows a fixed number of entities to chat in various conferences. In order to chat, the user is able to issue the following commands to the CPE:

Join nickname conference The user joins the named conference under the given nickname. A user participates in at most one conference at any time.

Datarequest messages All users in the conference receive this message.

Leave The user leaves the current conference.

There is a network through which the CPEs communicate. The interface from a CPE to the network is via a User Datagram Protocol (UDP). The CPE sends Protocol Data Units (PDUs) to the network. The network delivers these PDUs to the indicated CPE and adds the identification of the sender. There are no assumptions on the order of the arrival of the messages, nor on the reliability of the connection. A CPE can receive the following inputs from the network:

DataPDUin cpe message This CPE receives a messages from cpe.

AnswerPDUin cpe nickname conference The indicated cpe wants to join the named conference under the given nickname.

JoinPDUin cpe nickname conference Request to join the named conference from the indicated cpe under the supplied nickname.

LeavePDUin cpe The indicated cpe leaves the current conference.

To accomplish its task a CPE can send the following output messages. Only the last message is sent to the user; all other messages are directed to the indicated CPE via the network.

JoinPDUout cpe nickname conference Send a request to the named cpe to join the named conference. The network transforms this message to an AnswerPDUin input where the cpe of destination is replaced by the sender. Used to tell other CPEs that the user issues a Join.

AnswerPDUout cpe nickname conference Confirmation that cpe wants to participate in the conference. This is used as an answer to JoinPDUin.

DataPDUout cpe message Send the given message to the indicated cpe.

LeavePDUout cpe indicates to cpe that this user leaves the conference.

Data nickname message Show a received message to the user.

After the definition of appropriate data types to hold CPE identifiers, messages, nicknames and conferences, these messages are transformed directly to the corresponding algebraic data types. The state of a CPE is either Idle or it participates in a Conference. The list of tuples consisting of a CPEid and a Nickname records which other CPEs participate in this conference and their nicknames. This list is sorted and each CPE occurs at most once.

```
:: CPEstate = Idle | Conf ConferenceID Nickname [(CPEid,Nickname)]
```

The number of states is finite if the conference-ids, nicknames and CPEids are finite.

The specification for a given CPE is generated by the function in Fig. 8.1. It is sufficient to grasp the idea of the specification, so do not bother about all of the details. The occurring nicknames, conference-ids, and messages are modelled by simple algebraic datatypes. The lists of members of these types used (`Nicknames`, `ConferenceIDs`, `CPEids` and `Messages`) are generated by the systematic generation functions of G∀ST. For instance:

```
:: ConferenceID = Conference1 | Conference2

ConferenceIDs   :: [ConferenceID]
ConferenceIDs   =: generateAll pseudoRandomInts
```

The list of pseudo random integers is used by `generateAll` to control the order of values, see [15] for details.

All possible conferences occurring as state for a given CPE are generated by the function `Conferences::CPEid -> [CPEstate]`. Owing to the restrictions imposed on the list of participants (it should be ordered and each partner occurs at most once) it is not possible to use generic generation for the conferences.

Owing to the generic generation of lists of elements of a type, like `ConferenceIDs`, the generation function for the LTS, `CPElts`, remains correct if we add, change, or remove members in any of the types involved. Hence, it is more powerful and convenient to use than the definitions of the labelled transition systems used in most existing model-based test systems. For instance, TorX uses a specification of the LTS in Promela. In the Promela specification at [24] the number of partners is hardwired into the specification. Moreover, our specification is very concise if we compare it to all other specifications collected at [24]. The difference in size between this specification and the others is at least a factor of two.

8.3.3 Executing a Deterministic LTS

To use a given LTS as the basis for testing, we must be able to execute it. That is, given an LTS, a current state and an input we need to be able to determine the

```
CPElts :: CPEid -> LTS CPEstate CPEin CPEout
CPElts myId
 = { initial = Idle
   , trans
      = [   (Idle, Join nn confId
            ,[JoinPDUout cpe nn confId \\ cpe <- CPEids | cpe<>myId]
            , Conf confId nn [])
        \\  nn          <- Nicknames
        ,   confId      <- ConferenceIDs
        ] ++
        [   (conf, JoinPDUin cpe nn2 id
            ,[AnswerPDUout cpe nn id],Conf id nn (mkset (cpe,nn2) mem))
        \\  conf=:(Conf id nn mem)  <- Conferences myId
        ,   cpe         <- CPEids
        ,   nn2         <- Nicknames
        |   cpe <> myId && not (isMember cpe (map fst mem))
        ] ++
        [   (conf, AnswerPDUin cpe nn2 id
            ,[],Conf id nn (mkset (cpe,nn2) mem))
        \\  conf=:(Conf id nn mem)  <- Conferences myId
        ,   cpe         <- CPEids
        ,   nn2         <- Nicknames
        |   cpe<>myId && not (isMember cpe (map fst mem))
        ] ++
        [   (conf, Leave, [LeavePDUout cpe \\ (cpe,_) <- mem], Idle)
        \\  conf=:(Conf id nn mem)  <- Conferences myId
        ] ++
        [   (conf, LeavePDUin cpe, [], Conf c nn [t\\t<-mem|fst t<>cpe])
        \\  conf=:(Conf c nn mem) <- Conferences myId
        ,   (cpe,_)     <- mem
        ] ++
        [   (conf, DataPDUin cpe mes,[Data nn2 mes], conf)
        \\  conf=:(Conf id nn mem)  <- Conferences myId
        ,   mes         <- Messages
        ,   (cpe,nn2)   <- mem
        ] ++ // to compensate loss of AnswerPDU
        [   (conf, DataPDUin cpe mes,[JoinPDUout cpe nn id], conf)
        \\  conf=:(Conf id nn mem)  <- Conferences myId
        ,   mes         <- Messages
        ,   cpe         <- CPEids
        |   cpe <> myId && not (isMember cpe (map fst mem))
        ] ++
        [   (conf, Datareq mes,[DataPDUout cpe mes\\(cpe,_) <- mem],conf)
        \\  conf=:(Conf id nn mem)  <- Conferences myId
        ,   mes         <- Messages
        |   not (isEmpty mem)
        ]
   }
```

FIGURE 8.1 The specification of a CPE by the data structure LTS

associated output and new state. The realisation is very straightforward. Since the LTS is currently deterministic, we have in fact a finite state machine, FSM.

Often we prefer to give a sequence of inputs and obtain a list of associated outputs rather than giving a single input. This is achieved by the following function to execute a deterministic LTS.

```
runFSM :: (LTS s i o) [i] -> [[o]] | == s & == i
runFSM {trans,initial} inputs = run initial inputs
where
    run state [] = []
    run state [i:r]
     = case [(o,t) \\ (s,j,o,t) <- trans | s==state && i==j] of
        []       = [[]:run state r]  // undefined: ignore input
        [(o,t)] = [o :run t r]
        _        = abort "This LTS is not deterministic!"
```

8.3.4 The Implementation Under Test

We perform a *black box test* of the Implementation Under Test (IUT): we can only observe the output of the system given an input. To show clearly that a single input produces a sequence of outputs and a new state, we use the type:

```
:: IUT input output = IUT (input -> ([output],IUT input output))
```

It is often convenient to transform this to a function that converts a sequence of inputs to the associated outputs. This is done by:

```
runIUT :: (IUT i o) [i] -> [[o]]
runIUT iut [] = []
runIUT (IUT f) [a:r] = [o:runIUT iut r] where (o,iut) = f a
```

Here we define only the type of the IUT; it is all we need to know. In order to execute the test an implementation should be available.

8.3.5 Testing the Conference Protocol

After the introduction of a representation for model-based specifications and the tools to execute the specification and the IUT, we are ready to formulate properties to be tested automatically by G∀ST. We assume that an implementation of the CPE is available as a function of type `cpeImpl::CPEid -> IUT CPEin CPEout`.

A desirable property for any implementation of the conference protocol is that its outputs are equal to the outputs obtained by execution of the specification:

```
propCPE :: CPEid [CPEin] -> Bool
propCPE id input = runFSM (CPElts id) input == runIUT (cpeImpl id) input
```

This is a standard property for G∀ST. Hence, it is tested like any other property in G∀ST by executing `Start = test propCPE`.

This model-based property can be combined with an ordinary logical property. If we have a logical predicate `properState::CPEstate -> Bool` to check the sanity of states (CPEs are ordered and not duplicated), we can combine these properties to:

```
propCPEa :: CPEid [CPEin] -> Property
propCPEa id input = propCPE id input /\ (properState For (Conferences id))
```

When we are convinced that the protocol handles all CPEs equally, we can also limit the test to a single CPE-id. For `CPE1` the last property becomes:

```
propCPEb :: ([CPEin] -> Bool)
propCPEb = propCPEa CPE1
```

Testing these properties reveals some discrepancies between the initial versions of the specification and the implementation. The differences concern the handling of unusual inputs, like receiving a `DataPDUin` from a CPE that is not a member of the conference. This led us to improvements of the implementation as well as the specification. Afterwards G∀ST reports that these properties pass the tests.

When the implementation passes some significant number of tests it is tempting to believe that the implementation conforms to the specification. However,

analysis of the generated inputs showed that only a few conferences were established during the tests. Although the inputs are generated systematically, only a small fraction of the generated inputs correspond to actually entering a conference and sending messages. Typically, only one single data transfer within a conference is established in the first 100 tests that are generated.

Tests with systematically generated inputs appear to be very valuable to verify that the specification and the implementation ignore the same inputs, even if the sequence of messages is completely meaningless. This only tests that the IUT shows the specified behaviour: *robustness testing*.

8.3.6 Implementations with Other Types

The type of the IUT used above suits our tests very well. However, not every implementation we want to test has such a type. An alternative custom type for the implementation is `cpeImpl2::CPEid [CPEin] -> [[CPEout]]`. Even when the IUT produces a single stream of output tokens, `cpeImpl3::CPEid [CPEin] -> [CPEout]`, rather than a sequence of output per event, we can still test these implementations in G∀ST by adapting the property slightly:

```
propCPE' :: [CPEin] CPEid -> Bool
propCPE' input id = runFSM (CPElts id) input == cpeImpl2 id input

propCPE'' :: [CPEin] CPEid -> Bool
propCPE'' input id = flatten (runFSM (CPElts id) input) == cpeImpl3 id input
```

For `propCPE''` we only have lost the ability to check whether a particular output element is generated in response to the correct input. A particular element of the output might be generated too late or too early. Such a synchronization can cause serious troubles in the communication with a reactive system. In order to be able to detect these synchronization problems we prefer the somewhat more complicated type of output, `[[out]]`, above the plain list of output elements, `[out]`.

8.4 BETTER TEST DATA GENERATION FROM THE LTS

To check the correct behaviour for meaningful sequences of messages, *conformance*, we use the LTS as a source of information to produce meaningful input sequences. For instance, each meaningful sequence of inputs starts with an input corresponding to a transition from the initial state. We can use the existing knowledge of testing a FSM [25, 16]. An input sequence is usually called a path in the world of FSM-testing. If one assumes that the IUT is also deterministic, we do not learn anything new from executing a path which is a prefix of another tested path. If we furthermore assume that the IUT does not have more states than the specification, it is useless to test the same transition twice. Both assumptions are standard in FSM testing. We use this knowledge to construct a finite amount of longer and meaningful inputs. This implies that we are now able to prove things by exhaustive testing, instead of just executing successful tests. We discuss some test generation algorithms inspired by [21] and [25].

# CPEs	nick names	confer ences	mes sages	# states	trans itions	paths generated A1	A2	A3	A4
1	1	1	1	2	2	∞	1	1	1
2	1	1	1	3	9	∞	118	4	3
3	1	1	1	5	28	∞	>10,000	11	6
2	2	1	1	7	30	∞	>10,000	14	8
2	1	2	1	5	18	∞	27,848	7	6
2	1	1	2	3	12	∞	7,827	4	3
2	2	2	2	13	80	∞	>10,000	26	16
3	3	3	3	145	2070	∞	>10,000	567	282

TABLE 8.1 Number of paths generated for various size of types.

A1 From each state in the specification we only test the transitions from that state. To terminate each input sequence we randomly choose to end the path here or to use one of the possible transitions at each point. This is basically the algorithm for test-data generation used by TorX.

A2 Since it is useless to test the same transition twice, we terminate a path when there is no untested transition from the current state.

A3 The paths generated by the previous algorithm do not verify the final state at the end of the path. Since the IUT is a black box we cannot check this final state directly. The state can only be identified via the observed response to inputs. This algorithm checks the final state by performing additional transitions: we require that each transition occurs twice in the test suite.

A4 In this algorithm we use a function of type `state -> [input]`, to determine the inputs used to test the final state. Ideally, we use a *unique input output sequence*, UIO, or a *distinguishing sequence*, DS, to identify the final state [2]. Using a UIO we can verify whether we are in a given state by observing the output corresponding to the input sequence associated with that state. Using a DS we can identify the state by observing the output corresponding to an input sequence associated to the entire LTS. If the UIO and DS are unknown or do not exist, we can use a short sequence of inputs as an approximation.

Finding the shortest set of paths that achieve the goals of *A*3 and *A*4 is yet another variant of the travelling salesman problem. We use a simple algorithm that chooses the first transition available. An input sequence is terminated when we cannot extend it without taking a transition too often. Until all transitions are used enough we extend a prefix of one of the used inputs with transitions that still need to be done.

In Table 8.1 we list the number of states, the number of transitions in the LTS, and the generated number of input sequences according to algorithms above for various numbers of CPE's, nicknames, conferences and messages. By its nature *A*1 always generate infinitely many paths. For a particular test we choose some number of these paths. This table shows that the number of input sequences generated by *A*2 is rather big, even for specifications of modest size. In practice, it is too large for a quick and complete automatic test.

Algorithm *A*4 produces fewer paths and is more accurate, but requires known paths to verify the final state. For testing the conference protocol we used:

```
CPEtestSeq :: CPEstate -> [CPEin]
CPEtestSeq state = [ Datareq mess, Join nn confId ]
where mess = hd Messages; nn = hd Nicknames; confId = hd ConferenceIDs
```

By using the generic definitions for `Messages`, `Nicknames` and `ConferenceIDs` again, this definition is completely independent of the actual contents of these types.

Algorithm *A*3 is used when an appropriate test sequence for final states is not at hand. It usually gives good results.

It is important to realize that these tests only check if the IUT behaves as specified by the LTS; this is known as *conformance testing*. Testing with the generated input sequences does not show whether the IUT shows any unspecified behaviour. For this purpose we need exhaustive tests of all inputs in all states. The default generation algorithm of G∀ST for input sequences appears to test this effectively.

The algorithms *A*2..*A*4 are superior to a system where the test function decides dynamically whether it is useful to apply a given input. We do not have to wait until a suited input occurs. Moreover, we can decide easily when all states and transitions are visited and the testing is finished. This allows proofs of conformance instead of just successful tests.

8.5 FUNCTIONAL AND NONDETERMINISTIC SPECIFICATIONS

The `LTS` type straightforwardly represents labelled transition systems. However, it suffers from the following drawbacks:

1. It allows nondeterminism, but a thorough examination of the data structure is necessary to see whether the specification is deterministic or not.

2. It is limited to a finite number of transitions. Each and every state and input that can occur should be listed explicitly in the LTS. This makes it impossible to specify a system that echoes a given integer or string. It is desirable to use variables in states and functions.

3. It is impossible to use typical functional language features, like guards and pattern matching, in the specification.

All these problems are solved by using functions of type

```
:: Spec state input output :== state -> input -> [(state,[output])]
```

as specification. Just like above, we use implicit completion when we use this specification: inputs for states not specified do not change the state and produce no output. Consider the following system that returns the absolute value of every second negative integer. This small definition covers the transition for all integer lists.

```
cpeSpec myId Idle (Join nn conf)
 = [(Conf conf nn [],[JoinPDUout cpe nn conf \\ cpe<-CPEids| cpe<>myId])]
cpeSpec myId state=:(Conf conf nn mem) input
 # memberCPEs = map fst mem
 = case input of
    Datareq mes = [(state,[DataPDUout cpe mes \\ (cpe,_) <- mem])]
    Leave       = [(Idle,[LeavePDUout cpe \\ (cpe,_) <- mem])]
    DataPDUin id mes
      | isMember id memberCPEs
        = [(state,[Data nn mes\\(cpe,nn) <- mem | cpe == id])]
      | id<>myId
        = [(state,[JoinPDUout id nn conf])] // handle lost join
     AnswerPDUin id nn2 conf2
       | conf == conf2 && not (isMember id [myId: memberCPEs])
         = [(Conf conf nn (mkset (id,nn2) mem),[])]
     JoinPDUin id nn2 conf2
       | conf == conf2 && not (isMember id [myId: memberCPEs])
        = [(Conf conf nn (mkset (id,nn2) mem),[AnswerPDUout id nn conf])]
    LeavePDUin id = [(Conf conf nn [t \\ t=:(m,_) <- mem | m <> id],[])]
    _ = []          // to make the specification total
 cpeSpec _ _ _ = [] // to make the specification total
```

FIGURE 8.2 The specification of a CPE by a function

```
absoluteValue :: Spec Bool Int Int
absoluteValue b n
    | n<0
        | b = [(False, [~n])]
            = [(True , [])]
        = [] // other transitions are not allowed
```

To compare the new specification with the specification by a data structure in figure 8.1 we list the specification of the conference protocol by a function in figure 8.2. The second version is clearly more compact than the previous version using a data structure instead of a function. Since all lists yielded have at most length one, it is obvious that this specification is deterministic. In contrast to the specification by a data structure, listed in figure 8.1, this version also works if we use large (or infinite) domains like `Int` for cpe–ids and `String` for messages and nicknames. Using an infinite domain for a specification as used in figure 8.1 would result in an infinite representation of the specification, an specification by a function as in figure 8.2 can handle this without problems. This makes this kind of specifications really more powerful.

The test sequence generation algorithms, *A*1..*A*4, in section 8.4 operate on data structures. To uses these algorithms with functions as specifications we need to generate transitions from the specification by a function. For ordinary testing this is not needed. All transitions from a given state are produced by:

```
generateTrans :: (Spec s i o) s [i] -> [Transition s i o]
generateTrans spec s inputs = [(s,i,o,s2)\\i<-inputs, (s2,o)<-spec s i]
```

To obtain the entire transition relation, we just have to construct these transitions for every reachable state. For finite types we can use generic generation for the list of inputs to be tested. For infinite and extremely large types, like `Int`, the tester has to supply a list of inputs to be used.

8.6 TESTING NONDETERMINISTIC SYSTEMS

Until here we have assumed that each LTS is deterministic. Now we drop this assumption. An LTS is nondeterministic if there can occur several transitions for a given state and input. These transitions can differ in output and/or target state. Many real life systems contain some form of nondeterminism.

Consider a simple vending machine spcified by the nondeterministic LTS:

$$Final_T \xleftarrow{Coin/[Tea]} S_{tea} \xleftarrow{Button/[]} Idle \xrightarrow{Button/[]} S_{coffee} \xrightarrow{Coin/[Coffee]} Final_C$$

Initially the system is in the state *Idle*. If the button is pressed the machine decides to produce either tea or coffee, but nothing happens until a coin is inserted. A better vending machine returns to *Idle* after producing coffee or tea. From the input/ouput one cannot decide in which state the machine is after pressing the button. It is also impossible to guarantee that this machine is in state S_{tea} by supplying inputs, it is always possible for the machine to take the other branch. This machine is specified in G∀ST as:

```
vendingSpec Idle    Button = [(Stea,[]),(Scoffee,[])]
vendingSpec Stea    Coin   = [(FinalT,[Tea])]
vendingSpec Scoffee Coin   = [(FinalC,[Coffee])]
vendingSpec state   input  = []
```

To cope with this situation we use the **ioco**–test [22, 23, 4]. The name **ioco** stands for *input/output conformance*. The idea is that when an input belonging to the specification is supplied to the IUT, the observed output must be allowed by the specification. It is not required that all specified behaviour is implemented. When the specification contains a nondeterministic choice at some state for a given input, it is sufficient that at least one of these branches is implemented. This implies that an implementation with behaviour

$$Idle \xrightarrow{Button/[]} S_{coffee} \xrightarrow{Coin/[Coffee]} Final_C$$

is **ioco**–correct with respect to the specification above: any behaviour shown by this implementation is allowed by the specification.

The **ioco**–relation allows partial specifications: the implementation is allowed to respond to inputs not occurring in the specification. Due to the restriction that inputs should belong to the specification, this additional behaviour is not considered in the **ioco**-correctness. For instance the vending machine that produces drinking chocolate after being hit, the input *Bang*, and the insertion of a coin is an **ioco**–correct implementation of the specification above.

$$Final_C \xleftarrow{Coin/[Cacao]} S_{cacao} \xleftarrow{Bang/[]} Idle \xrightarrow{Button/[]} S_{coffee} \xrightarrow{Coin/[Coffee]} Final_C$$

An implementation that can offer cacao after pushing the button and inserting a coin, however, is incorrect.

$$Final_C \xleftarrow{Coin/[Cacao]} S_{cacao} \xleftarrow{Button/[]} Idle \xrightarrow{Button/[]} S_{coffee} \xrightarrow{Coin/[Coffee]} Final_C$$

The output *Cacao* after inputs belonging to the specification, *Button* and *Coin*, is not allowed by the specification. This error is discovered during testing as soon as the implementation produces cacao for the first time.

During the test we do not know always in which state of the specification we are currently. For instance, after applying the input *button* and observing that there is no output, the implementation might be in a state corresponding to S_{tea} or to S_{coffee}. To deal with this nondeterminism we maintain a list of possible current states, instead of a single current state. After the input `Button` in the state `Idle` the list of possible states is `[Stea,Scoffee]`.

This is implemented by `testIoco`. Similar to `test` this function yields a report encoded in a list of strings. For clarity we use a separate function `testIoco` rather than a new operator for `test`.

```
testIOCO :: (Spec s i o) [s] (IUT i o) [[i]] -> [String] | == o
testIOCO spec states iut paths = test 1 paths
where
  test n []        = ["All tests successful"]
  test n [p:paths] = [toString n: ioco iut states p (test (n+1) paths)]
  ioco iut       []     path     cont = ["Error!"]
  ioco iut       states []       cont = ["OK\n":cont]
  ioco (IUT iut) states [i:path] cont = ioco iut2 states2 path cont
  where (iutout,iut2) = iut i
        states2 = [t\\s<-states, (t,specout)<-spec s i| specout==iutout]
```

This test does not require that the system is really nondeterministic. It can, for instance, be used to test the conference protocol where the input is generated by one of the algorithms discussed above. Paths can be generated by the algorithms *A*1..*A*4, introduced in section 8.4. The needed `start` function is: `Start = testIOCO (cpeSpec CPE1) [Idle] (cpeImpl CPE1) (A4 (CPElts CPE1) CPEtestSeq)`.

A more sophisticated **ioco**–test algorithm might generate the input on basis of the observed behaviour. This *on the fly* testing [9] remains future work.

Note that this **ioco**–test is done by a small function inside the G∀ST framework. All other test systems for model based specifications (like TorX) are huge stand alone systems. These systems lacks the abilities to generate data types G∀ST has and have troubles with properties of these data types.

8.7 RELATED WORK

The closest related test system for logical properties (i.e. the original G∀ST) is QuickCheck [7, 8]. The discriminating difference between QuickCheck and G∀ST is the systematic test data generation in G∀ST. Test data generation in QuickCheck is based on a class, the user has to supply an instance for each new type, and random data generation. In G∀ST the test data generation for a new type comes for free since it is based on generics [3, 11]. Moreover, the generation of test data is systematic from small to large without duplicates. When a property holds for all values in a type, it is proven.

With the extension of G∀ST introduced in this paper makes it a model based test system [6] like TorX [22, 21], Autolink [19, 14], TGV [**?**], and UIO Test [9]. Basically these systems generate inputs for the system to be tested based on

the LTS–specification. Currently these systems have difficulties with conditions on values and the generation of these values. In G∀ST however, such conditions can easily be expressed in first order logic. We are aware of a number of running projects to extend model based test systems with capabilities to handle restrictions on types. No results have been reported yet. The model based specifications in CLEAN appear to be clearer, shorter and more general than the example specifications collected at [24].

In [8] it is shown how Quickcheck can handle systems with a state. These systems are monad based, and specified in logic instead of an LTS. We expect that those extensions can be incorporated into G∀ST, and that Quickcheck can be extended with the capabilities of G∀ST.

8.8 CONCLUSION

In this paper we extended G∀ST with the ability to test software described by model–based specifications. We used labelled transition systems for these specifications, and shown that such an LTS can be better specified by a function than data type. The well–know **ioco**-relation for nondeterministic systems can be tested by a small extension to the test library G∀ST, instead of a huge stand alone test system.

By representing a labelled transition system as a data type and enabling the execution of such an LTS, we are able to test systems specified by an LTS in G∀ST. This is a significant improvement since many interesting systems are specified by a model instead of a property in first order logic.

Moreover, such an LTS is used as a basis for test data generation. These input sequences test that the system behaves correct for inputs that are part of the specification. The default data generation of G∀ST is used to verify that the system does not show undesired behaviour for other inputs.

The use of functions instead of a data type to specify an LTS has two significant advantages. The specification becomes even more concise and it is able to handle infinite data types for labels and states.

The model based testing is well integrated with the automatic testing of logical properties. This makes G∀ST with this extension stronger than existing model based testers. These systems are known to be weak at testing data types. There are several projects running to extend model based test systems with the ability to generate data values, no results have been reported yet.

Acknowledgement

We thank Peter Achten, Marko van Eekelen, Jan Tretmans, Rene de Vries, Arjen van Weelden and Ronny Wichers Schreur for their contributions to this paper.

REFERENCES

[1] Douglas Adams. *The Hitch Hiker's Guide to the Galaxy*, ISBN 345391802, 1979.

[2] A. Aho, A. Dahbura, D. Lee, and M. Uyari *An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours* In Protocol Specification, Testing and Verification VIII, volume 8, 1998.

[3] A. Alimarine, R. Plasmeijer. *A Generic Programming Extension for Clean*. IFL2001, LNCS 2312, pp.168–185, 2001.

[4] M. van der Bijl, A. Rensink, and J. Tretmans *Component Based Testing with IOCO*, CTIT Technical Report TRCTIT0334, University of Twente, 2003.

[5] A. Belinfante, J. Feenstra, R. Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, L. Heerink *Formal Test Automation: A Simple Experiment*, in *Int. Workshop on Testing of Communicating Systems 12* pp 179-196, 1999.

[6] E. Brinksma, J. Tretmans *Testing Transition Systems: An Annotated Bibliography*", in *Modeling and Verification of Parallel Processes–4th Summer School MOVEP 2000* LNCS 2067, pp 186-195, 2001.

[7] K. Claessen, J. Hughes. *QuickCheck: A lightweight Tool for Random Testing of Hasskell Programs*. International Conference on Functional Programming, ACM, pp 268–279, 2000. See also `www.cs.chalmers.se/~rjmh/QuickCheck`.

[8] K. Claessen, J. Hughes. *Testing Monadic Code with QuickCheck*, Proceedings of the ACM SIGPLAN workshop on Haskell 2002, Pittsburgh, pp 65–77, 2002.

[9] J. Fernandez, C. Jard, T. Jéron, C. Viho *Using On-the-Fly Verification Techniques for the generation of test suites*, Conference on Computer Aided Verification, LNCS 1102, 1996.

[10] L. Heerink, J. Feenstra, and J. Tretmans *Formal Test Automation: The Conference Protocol with PHACT* In H. Ural et al *Testing of Communicating Systems - Procs. of TestCom 2000*, Kluwer, pp 211–220, 2000.

[11] R. Hinze, *Polytypic values possess polykinded types*, Fifth International Conference on Mathematics of Program Construction, LNCS 1837, pp 2–27, 2000.

[12] Gerard J. Holzmann *SPIN Model Checker, The: Primer and Reference Manual* Addison Wesley, isbn 0-321-22862-6, 2003.

[13] N. Goga *Comparing TorX, Autolink, TGV and UIO Test Algorithms*, SDL 2001: Meeting UML: 10th International SDL Forum Copenhagen, Denmark, June 27-29, 2001, Proceedings. LNCS 2078, pp 379–402, 2001.

[14] A. Kerbrat, T. Jéron, R. Groz *Automated Test Generation from SDL Specifications*, in *The Next Millennium–Proceedings of the 9th SDL Forum*, pp 135–152, 1999.

[15] Pieter Koopman, Artem Alimarine, Jan Tretmans and Rinus Plasmeijer: *Gast: Generic Automated Software Testing*, in Ricardo Peña: *IFL 2002, Implementation of Functional Programming Languages*, LNCS 2670, pp 84–100, 2002.

[16] D. Lee, and M. Yannakakis, M *Principles and Methods for Testing Finite State Machines– A Survey*, The Proceedings of the IEEE, 84(8), pp 1090-1123, 1996.

[17] George Mealy *A method for synthesizing sequential circuits*, Bell System Technical Journal, 34(5):1045–1079, 1955

[18] Rinus Plasmeijer and Marko van Eekelen: *Concurrent Clean Language Report (version 2.0)*, 2002. `www.cs.kun.nl/~clean`.

[19] M. Schmitt, A. Ek, J. Grabowski, D. Hogrefe, and B. Kock *Autolink – putting SDL–based test generation into practise* Proceedings of the 11th International Workshop on Testing Communication Systems, pp 227–243, Kluwer Academic, 1998.

[20] Mike Spivey *The Z Notation: A Reference Manual,* 2nd *ed*, Prentice Hall, 1992.

[21] J. Tretmans, E. Brinksma *Côte de Resyste – Automated model–based Testing*, in *Progress 2002 –* 3rd *Workshop on Embedded Systems*, pp 246–255, 2002.

[22] J. Tretmans *Test generation with inputs, outputs and repetive quiscence*. Software–Concepts and Tools, 17(3):103–120, 1996.

[23] J. Tretmans *Testing concurrent systems: A fromal approach*. In J. Baeten and S. Mauw *Concur'99* LNCS 1664, pp 46–65, 1999.

[24] Various formal specifications of the conference protocol. `http://fmt.cs.utwente.nl/ConfCase/v1.00/specifications/specs.html`

[25] H. Ural, *Formal methods for test sequence generation*, Computer Communications Journal, **15**(5), pp 311–325, 1992

This book collects the latest research developments in the use of functional programming languages. The contents highlight major research goals and engineering concerns in the subject including:

- real-time and resource-bounded functional programming
- connections between static analysis methods and functional programming
- implementation of mobile code functional languages
- automated testing of application programs and system models

These research contributions are drawn from the international symposium on Trends in Functional Programming.
This meeting provides a vital forum for the dissemination of the latest technical advances in the field.

Dr Stephen Gilmore is a Senior Lecturer in the Laboratory for Foundations of Computer Science at The University of Edinburgh. His research interests include the definition, development and use of functional programming languages. He has previously edited the second volume in this series.

ISSN 1743-4505 (Print)
ISBN 1-84150-122-0

intellect
PO Box 862
Bristol BS99 1DE
United Kingdom
www.intellectbooks.com