Workshop on
Functional Programming in the Real World

Organizers:

Robert Giegerich, Universität Bielefeld

John Hughes, Chalmers University of Technology

May 16-20, 1994

Compiler technology for functional languages has reached the point where it is feasible to program some medium to large applications in a purely functional language, and indeed efforts are already underway in many areas. There is much to be learned from using functional languages in real situations — on the one hand, how large functional programs should be designed, and on the other, how existing languages, implementations and tools need to be improved.

Before this background, the workshop brought together users of functional languages in ambitious software projects, and a small group of language designers and implementors. The workshop demonstrated the scope of applications of functional programming to date. Users exchanged their experiences and provided feedback to language designers and implementors. Discussions evaluated the state of the art and outlined central problem areas for the further development of functional programming. The workshop focussed on the following topics:

- Hardware Description and Simulation,
- Real-Time Systems,
- Bioinformatics,
- Communication Protocols,
- Scientific Computing,
- Programming Languages and Systems,
- Software Tools,
- Prototyping,
- User Interfaces.

Besides the presentation of applications, there were reports on recent developments in functional language implementations (SML, Haskell, Clean). An evening discussion on "Functional Programming in Education" revealed that functional

languages are about to become an important vehicle for teaching computer science — no longer restricted to introductory programming courses, or functional programming per se. A more comprehensive summary of this workshop is planned to appear in the Journal of Functional Programming.

The workshop was generally perceived by the participants as an outstanding opportunity for exchange of ideas and experiences. A number of spontaneous experiments were performed cooperatively, using the Dagstuhl computing facilities. We are especially grateful to the Dagstuhl team for their professional support for planning and conducting the workshop.

Monday, May 16

Session 1: Hardware Description and Simulation

Simon Finn	*The LAMBDA Experience*
John O'Donnell	*Hydra: Haskell as a Computer Hardware Description Language*
Werner Pohlmann	*Discrete Event Simulation via Functional Programming*

Session 2: Real-Time Systems

Markus Freericks	*Developing a Language for Real-Time Numeric Applications*
Staffan Truve	*Declarative Real-Time Systems*
Mike Williams	*Erlang - A Functional Programming Language for Developing Real-Time Products*
Hugh Glaser and Pieter H. Hartel	*The Resource Constrained Shortest Path Problem implemented in a Lazy Functional Language*

Tuesday, May 17

Session 3: Programming Languages and Systems

Martin Alt	*Practical Experience using a Nondeterministic Functional Language*
Jasper Kamperman	*GEL, a Graph Exchange Language*
Richard Frost	*W/AGE: The Windsor Attribute Grammar Programming Environment*

Session 4: Biological Sciences

Marc Feeley and Marcel Turcotte	*A Parallel Functional Program for Searching a Discrete Space of Nucleic Acid 3D Structures*
Robert Giegerich	*Towards a Declarative Pattern Matching System for Biosequence Analysis*
Stefan Kurtz	*Fundamental Algorithms for Comparing Biosequences*
Patrick J. Miller	*Rational Drug Design in Sisal '90*

Wednesday, May 18

Session 5: Tools

Peter Thiemann	*Towards Software Tools in Haskell*
Peter Baumann	*Building Software Maintenance Tools in Standard ML*
H.R. Walters	*Implementing Tools by Algebraic Specification*
Einar Wolfgang Karlsen	*Tool Integration in a Persistent and Concurrent Functional Setting*

Thursday, May 19

Session 6: Prototyping

Hugh Glaser	*Prototype Development in a Lazy Functional Language*
Pieter H. Hartel	*Prototyping a Smart Card Operating System in a Lazy Functional Language*
Cordelia Hall	*Natural Expert: A Commercial Functional Programming Environment*
Rex Page	*Fortran for Functional Programming: Is that absurd or what?*

Session 7: User Interfaces

David Goblirsch	*Training Hidden Markov Models using Haskell*
Rick Morgan	*The LOLITA System*
Magnus Carlsson	*Client/Server Programming with Fudgets*
Marko van Eekelen	*A Lazy Functional Spreadsheet written in a Lazy Functional Language*

Friday, May 20

Session 8: Communication Protocols and Scientific Computing

Edoardo Biagioni	*Theory and Practice: the Fox Net*
Peter Lee	*Advanced Development of Systems Software*
J.A. Sharp	*Functional Programming Applied To Numerical Problems*
Hugh Glaser	*Some Lattice-based Scientific Problems, Expressed in Haskell*

Practical Experience using a Nondeterministic Functional Language

Martin Alt, Universität des Saarlandes

Trafola is an advanced functional programming language and system. Originally designed and developed as a tool for the specification and execution of complex tree transformations, it possesses sophisticated search and replacement mechanisms. It includes higher order functions, nondeterministic pattern constructs and second order predicates but is strict. For some practical relevant problems, the language has been shown as very useful. We show the benefits using such an expressive language for real programming a parser generator.

Building Software Maintenance Tools in Standard ML

Peter Baumann, Universität Zürich

The aim of the project AEMES (An Extensible Maintenance Engineering System) is the design and the implementation of a prototype of an extensible reverse engineering environment for real world COBOL-74 applications. Software maintenance is an important and costly part of the software life-cycle, but it is difficult and time-consuming. Our approach is to use modern techniques such as functional programming in order to build a flexible and powerful environment.

The two major design principles are:

- to keep the architecture and the tools of the environment generic and adaptable to other dialects of COBOL or even to other programming languages
- to build an extensible environment such that new tools can be integrated without having to redefine and re-implement the existing system

We conducted a feasibility study in order to decide whether we can use Standard ML as implementation language. During this study, in which we used Poly/ML[1], we were able to build a scanner/parser, a program browser and a cross referencer for COBOL-74. There are many different variants of COBOL dialects and therefore there exists no universally valid grammar. The sheer size of the syntax complicates the parsing of the language even more. To cope with this problem, we developed a two-step approach for parsing programs. In the first step, we split a program into coherent program parts, such as divisions, sections, and paragraphs. In the second step, single statements are scanned. The browser and the cross referencer combine the information of the decomposition step with the information gained in the scanning process. The browser allows fast navigation in programs. The cross referencer generates a table which contains each token of

[1] Poly/ML is a trademark of Abstract Hardware Limited

a program together with the information about all positions where the symbol occurs. Because the browser and the cross referencer can be used interactively, we put graphical user interfaces on top of them.

This study showed that it is indeed possible to implement software analysis tools with acceptable performance for real-world programs in Standard ML. Especially, the power of higher-order functions and the polymorphic type system made it possible to implement the tools in comparably short time. It is worth noting that only one team member knew Standard ML before we started the study. Nevertheless, we were able to realize full functional tools within only four months. The main problem we encountered in using Standard ML was the lack of adequate connections to state-of-the-art user interface technologies and the missing support for persistent objects. But the overall results were encouraging, so we decided to continue to work with Standard ML.

Theory and Practice: the Fox Net

Edoardo Biagioni, Carnegie Mellon University

We have used an extension of SML/NJ to build a complete implementation of the TCP/IP standard suite of network protocols.

The implementation is structured according to well-understood principles from the networking field, but we have been able to use SML's modules to give these principles concrete expression. The implementation of TCP sequentially interprets commands placed asynchronously onto a per-connection queue; this quasi-synchronous structure gives an implementation that is simpler than its fully-asynchronous counterparts.

We use the callcc extension to SML to implement coroutines, byte arrays to represent network packets, and 8, 16, and 32-bit integers to compute and store header fields and checksums. Except for the direct device access routines, all of the code is safe in the sense that bounds are checked on all array accesses and type casting operations are not used.

The performance of the implementation is within a factor of two of a comparable C implementation for most protocols; the TCP protocol is complex and the C implementation is highly optimized, so the relative performance is not as good. We find that our implementation is highly maintainable, with few of the common problems that plague systems programs such as memory misallocation and illegal pointer references.

Client/Server Programming with Fudgets

Magnus Carlsson, Chalmers University of Technology

The Fudget library, which is based on parallel stream processors, has been used to write Haskell/LML programs with graphical user interfaces. In collaboration with Thomas Hallgren, it has now been enhanced with capabilities for UNIX socket communication, making it possible to write both client and server applications with Fudgets. Writing servers which are capable of handling many clients simultaneously seems to be a new application area for Fudgets.

I'll describe the small client/server application Chat, where the server broadcasts messages from connected clients to all clients.

A Functional Spreadsheet

Marko van Eekelen with Walter de Hoon, University of Nijmegen

Recent developments in research on efficiency of code generation and on graphical input/output interfacing have made it possible to use a pure, lazy functional language to write efficient programs that can compete with industrial applications written in a traditional language.

Ongoing work is described concerning the development of a spreadsheet written in the pure lazy functional language Clean which offers facilities to achieve state-of-the-art performance using uniqueness typing. The design was geared towards combining existing programs as much as possible (e.g. a theorem prover as spreadsheet language interpreter and an editor for the definition of new functions).

An interesting aspect of the developed spreadsheet-prototype is that the language with which the user specifies the relations between the cells of the spreadsheet is not one of the standard spreadsheet languages but also a pure lazy functional language. Another interesting aspect of this spreadsheet is that expressions in the spreadsheet language are evaluated not by a standard interpreter but by a theorem prover which gives the possibility to incorporate cells with expressions proving properties of certain expressions within the spreadsheet. Spreadsheets made in such a way are highly reliable and relatively easy to understand and to reason about.

Future work is directed not only towards finishing the implementation and improving the efficiency of the theorem prover to the level of efficiency of standard functional language interpreters but also towards extensions to a multi-user distributed version.

A Parallel Functional Program for Searching a Discrete Space of Nucleic Acid 3D Structures

Marc Feeley and Marcel Turcotte, Université de Montréal

We will present the *Nucleic Acid 3D structure problem*(NA3D) and a constraint satisfaction algorithm used to search a discrete space of molecular conformations. Most approaches to this problem are based on numerical methods; we have adopted a more symbolic approach. For this reason and to ease experimentation, we have implemented our system (MC-SYM) using the Miranda functional programming language.

As the size of the problems processed by MC-SYM increased (several days to solve on a high-performance workstation) it became clear that we needed to increase its speed. We report here our experience in improving the system's performance, especially regarding its parallelization with the Multilisp language. Using two realistic data sets, we compare the original sequential version of the program written in Miranda to the new sequential and parallel versions written in C, Scheme, and Multilisp, and explain how these new versions were designed to attain good absolute performance. Critical issues were: the performance of floating-point operations, garbage collection, load balancing, and contention for shared data. We found that speedup was dependent on the data set. For the first data set, nearly linear speedup was observed for up to 64 processors whereas for the second the speedup was limited to a factor of 16.

The LAMBDA Experience

Simon Finn, Abstract Hardware Limited

Abstract Hardware Limited (AHL) was formed to produce and sell formally-based tools for digital hardware design. We chose to construct these tools – the LAMBDA system – in Standard ML using (mostly) functional programming techniques. The LAMBDA system is now 7 years old and contains more than 170K lines of code, making it one of the world's largest functional programs.

This talk will discuss the consequences of the decision to use SML:

- Do the well-publicised *features* of functional programming (polymorphism, higher-order functions, *:::*) actually produce real-world *benefits*?
- Do these benefits out-weigh the real penalties (lack of mature implementations, problems interfacing to C-based system, *:::*) of using functional programming?

- What effects do the non-functional parts of LAMBDA have on the efficiency, reliability and maintainability of the total system?
- Why did AHL become an SML vendor?

In short, I shall address the question 'if functional programming is so good, why does (almost) nobody use it?'.

Developing a Language for Real-Time Numeric Applications

Markus Freericks, TU Berlin

The language Aldisp is presented, and the special problems encountered both in its design and implementation described. Aldisp is targetted at the design and implementation of digital signal processing (DSP) applications with control components. Besides the synchronous data-flow model, Aldisp supports concepts of asynchronous communication, real-time, and I/O. A wide range of numerical behaviour can be specified.

Aldisp is a quasi-functional language: it supports higher- order functions, dynamic typing, and a garbage-collected heap; and no "obvious" imperative features are included. On the other side, I/O and time have to be modelled. This is done by introducing a two-level concept of computation: scheduling and I/O are dependent on global state and side-effects, but the "computational" part of the language is purely functional.

The design of Aldisp was done without access to a running implementation, and without much regard to problems of efficiency. In that respect, Aldisp is a specification language that enables the succinct and precise description of real-time numeric algorithms.

In the last years, a compiler for Aldisp has been written. During its development, many thousand lines of code have been written and discarded, the implementation language has changed (from Scheme to SML/NJ), and many of the basic ideas of how the compiler should work have changed. At the moment, its centerpiece is an on-line partial evaluator developed from an abstract interpreter and an abstract scheduler.

The talk will center on the unique aspects of Aldisp and their domain-specific motivation. Some of these language features have been shown to create more problems than they solve; others fulfill their tasks, but are overly complex to implement.

W/AGE: The Windsor Attribute Grammar Programming Environment

Richard Frost, University of Windsor

W/AGE is a programming environment that enables language processors to be constructed as executable specifications of attribute grammars. W/AGE has been under development since 1988 and has been used in a number of projects including the development of natural language interfaces, SQL processors, theorem provers and circuit design transformers within a VLSI design package.

The W/AGE programming language consists of a set of higher-order functions that extend the standard environment of a pure lazy functional programming language. The notation is similar to textbook notation for attribute grammars.

Each grammar production in a W/AGE program is implemented as a syntax-directed evaluator using a functional variation of classical top-down parsing with full backtracking. Fully general attribute dependencies, including "inheritance from the right" are possible owing to the normal-order evaluation strategy employed by the lazy functional host language. Left-recursive productions are accommodated using a novel technique which involves the use of recognizers that act as guards. Polynomial complexity for recognition of ambiguous languages is achieved through memoization at the source-code level.

Towards a Declarative Pattern Matching System for Biosequence Analysis

Robert Giegerich, Universität Bielefeld

A number of approaches have aimed to provide flexible tools for biosequence analysis. Efficient algorithms are known for isolated standard problems (alignment, fast similarity search, approximate matching, repeats and palindromes, cloverleaf structures, ...). Our goal is to embed these into a pattern matching system useful to the working biologist. One way to achieve this is a pattern matching language with a declarative semantics. Besides describing complex patterns in a transparent way, a user must also be able to integrate own functionality with built-in features. Thus abstractness, flexibility and extensibility are the main requirements of such a system. A prototype system was implemented via a combinator language that closely minimizes the pattern language of Mehldau's dissertation. A large number of pattern matching algorithms have since been reviewed with respect to their suitabilty as the basic machine for a more efficient implementation. This talk concentrates on the motivation and the language design aspects.

Prototype Development in a Lazy Functional Language

Hugh Glaser with Celia Glass, University of Southampton

This presentation reports on the development of an algorithm in a lazy functional language.

The development takes place as a dialogue between the consultant in the applications domain (Operational Research), the Consultant, and the applications programmer, the Programmer. This dialogue required a number of iterations and prototypes, and many of the facilities peculiar to functional languages were used in support of this process. In particular, a number of data type transformations and generalisations were performed which meant that the final program corresponded more closely to the Consultant's view of the problem than the original capture.

Some Lattice-based Scientific Problems, Expressed in Haskell

Hugh Glaser with Bryan Carpenter, University of Southampton

This presentation explores the application of a lazy functional language, Haskell, to a series of grid-based scientific problems—solution of the Poisson equation and Monte Carlo simulation of two theoretical models from statistical and particle physics. The implementations introduce certain abstractions of grid topology, making extensive use of the polymorphic features of Haskell. Updating is expressed naturally through use of infinite lists, exploiting the laziness of the language. Evolution of systems is represented by arrays of interacting streams.

The Resource Constrained Shortest Path Problem implemented in a Lazy Functional Language

Hugh Glaser, University of Southampton and
Pieter H. Hartel, University of Amsterdam

The resource constrained shortest path problem is an NP-complete problem for which many ingenious algorithms have been developed. These algorithms are usually implemented in FORTRAN or another imperative programming language. We have implemented some of the simpler algorithms in a lazy functional language. Benefits accrue in the software engineering of the implementations. Our implementations have been applied to a standard benchmark of data files, which is available from the Operational Research library of Imperial College, London. The performance of the lazy functional implementations, even with the comparatively simple algorithms that we have used, is competitive with a sophisticated FORTRAN implementation.

Training Hidden Markov Models using Haskell

David M. Goblirsch, The MITRE Corporation

First I will briefly outline the structure of phonetically based automatic speech recognition systems. Then I will describe how Hidden Markov Models (HMMs) are used to link the acoustic signal and the basic sound units, called phones, that are used to describe word pronunciations. I will describe how the Viterbi dynamic programming algorithm is used to align the phonetic representation of an utterance with the acoustic signal, and how the results of this alignment can be used to reestimate the HMM parameters. Finally, I will give an overview of a suite of programs used to train HMMs from a corpus of speech training data.

Natural Expert: A Commercial Functional Programming Environment

Cordelia Hall, University of Glasgow

Natural Expert is a product that allows users to build knowledge based systems. It uses a lazy functional language, Natural Expert Language, to implement backward chaining and provide a reliable knowledge processing environment in which development can take place. Customers from all over the world buy the system and have used it to handle a variety of problems, including applications such as airplane servicing and bank loan assessment. Some of these are used 10,000 times or more per month.

Prototyping a Smart Card Operating System in a Lazy Functional Language

Pieter H. Hartel, University of Amsterdam

The operating system of a smart card is an example of a mission critical system. Distributed in millions, smart cards with their small 8-bit CPU support applications where values are transferred only protected by the strength of a cryptographic protocol. This strength goes no further than the implementation of the software in the card and terminal allows. Because of its complexity, to guarantee absolute reliability of the smart card software is prohibitively expensive. Obtaining a high level of confidence in the implementation of a smart card application is essential for the widespread acceptance of smart cards. A highly structured design of the smart card operating system gives the designer control over the complexity of the system.

A functional language has been used to prototype a smart card operating system. The prototype has the same structure as the real operating system and it offers most of the functionality of the real system. The well defined semantics of pure

functional languages and their compositionality in particular are instrumental to the structuring of the prototype. With the functional language implementation as reference the reliability of implementation can be assessed at detailed level.

GEL, a Graph Exchange Language

Jasper Kamperman, CWI, Amsterdam

Graph-structured data types play a role in a large variety of complex software systems. Especially software performing symbolic manipulation, such as a compiler or a symbolic algebra system (graph rewriting), makes use of this kind of data types. Several trends, e.g. the growth of distributed computing and the integration of software developed for different purposes, cause a demand for the efficient, language-independent, exchange of graph-structured data.

GEL can be characterized by observing how it extends the capabilities of other formalisms for external data representation:

- GEL contains a dynamic abbreviation mechanism, which allows the use of very short identifiers for types and attribute names.
- Instead of labels to identify shared subgraphs or circularities, GEL has relative indices. This is comparable in advantages to the use of de Bruijn indices in the lambda calculus.
- The semantics of GEL assumes the existence of a stack of subgraphs. This leads to an efficient implementation for the important class of DAGs (Directed Acyclic Graphs).
- GEL is compositional: if a graph is composed of several subgraphs, its GEL text can be composed of the GEL texts of its subgraphs.

As a result, typical GEL representations of large tree-like graph structures asymptotically require an average of one byte storage for representing a node in the graph.

We claim that in many cases, GEL removes the need to implement a more efficient exchange protocol for production versions of a system. Thus, components in the prototype phase can be mixed freely with production versions of other components.

An experimental implementation of GEL has been used satisfyingly for the interfacing between software components generated from algebraic specifications and components built in a more traditional way.

Tool Integration in a Persistent and Concurrent Functional Setting

Einar Wolfgang Karlsen, Universität Bremen

Integration of individual CASE tools into a complete SDE require control and data integration. We present a framework for a common tool environment in a persistent and concurrent functional setting. The persistency features ensure type safe access to the persistent store for first order as well as higher order values. The concurrency model is related to the higher order pi-Calculus and offer features in support of process creation, selective communication, distributed programming and dynamic connectivity.

We then present the framework for control integration. It is demonstrated how a persistent setting offers the combined advantages of the tightly and loosely coupled integration paradigms in terms of type-safety of the former and flexibility of the latter. We then present the framework for tool encapsulation, and show how it offers significant improvements over existing technologies. We round off the discussion on control integration techniques by presenting a higher order message broadcast server. Also in this context, several important improvements over traditional imperative frameworks can be demonstrated.

We briefly discuss the architecture of the functional DBMS based on a paradigm where processes are used to encapsulate state. The DBMS is here viewed as a stream processing agent taking transaction functions as parameters and delivering answers as result. We present the approach to transaction development based on the state transformer monad, and discuss the modelling of a few transaction schemes. Future research topics are finally presented.

Fundamental Algorithms for Comparing Biosequences

Stefan Kurtz, Universität Bielefeld

Molecular biologists frequently compare biosequences to see if any similarities can be found in the hope that what is true of one sequence either physically or functionally is true of its analogue. Such comparisons are made in a variety of ways, some via rigorous algorithms, other by manual means, and other by a combination of these two extremes. The most established technique for sequence comparison is dynamic programming. It is useful for the basic problems as well as for their variations, where it is often used together with suffix trees, a kind of inverted index of a sequence. In our talk we review dynamic programming and suffix trees from the viewpoint of functional programming. Using programming with unknowns we describe an implementation of dynamic programming in a lazy functional language. Using higher order functions we show how to unify the concept of suffix trees, which come in various flavours – with different degrees of compactness, auxiliary information to help with their construction, and specific

annotation according to their intended use. We describe a new, "lazy" suffix tree construction, which is much simpler and even faster than the well-known constructions. A main goal of the talk is to show that our functional implementation of dynamic programming and the lazy suffix tree construction can serve as the building blocks of a system, that solves a variety of sequence comparison problems.

Advanced Development of Systems Software

Peter Lee with Robert Harper, Carnegie Mellon University

The long-term objectives of the Carnegie Mellon Fox Project are to improve the design and construction of systems software and to further the development of advanced programming languages. We have been using an extension of the Standard ML programming language in the design and construction of systems software, in particular a suite of network communications protocols. Many of the key aspects of the design of the implementation are taken from the x-kernel system, which find natural expression in the functional-programming paradigm, especially in the types and the use of higher-order functions and continuations. The modules system is especially useful for structuring the layering of protocols and providing a useful framework for conditional compilation and testing. Current performance figures, while not yet competitive with implementations in C, are quite encouraging and are expected to improve dramatically. This Project involves several faculty members and spans a wide range of research areas, from experimental development of systems software to advanced compiler development to language design.

Rational Drug Design in Sisal '90

Patrick J. Miller, University of California, Lawrence Livermore National Lab.

Sisal '90 is a general purpose, applicative language targeted at developers of large, scientific codes. Its semantics are completely functional so determinate behavior is guarantied. The language is built around features common to many scientific codes: arrays, if-then-else's, and loops. We have successfully implemented many scientific kernels in Sisal. Their runtime performance is often better than parallelized, vectorized FORTRAN. The efficiency develops from the careful choice of Sisal language features, aggressive optimizations, and copy eliminations. We will discuss our experiences developing a rational drug design code (RDD) in Sisal '90. The RDD's kernel iterates over a force computation. The force computations can be done in parallel, but the forces must then be accumulated in a histogram – a traditionally tricky problem for functional languages. Sisal '90 offers new features that help solve this and other problems for scientific code developers.

LOLITA: A natural language processing system written in Haskell

Rick Morgan, Computer Science Labs. (SECS), Durham

The LOLITA system is a state of the art natural language engineering system, able to grammatically parse, semantically and pragmatically analyse, reason about and answer queries on normal complex English texts such as articles from the financial pages of quality newspapers.

It is written in Haskell and currently stands at over 35,000 lines of source code (excluding comments). The semantic network, which is the systems central data structure, contains over 30,000 nodes (soon to be increased to over 120,000), allowing more than 100,000 inflected word forms. It presently runs on a Sparc workstation with a minimum of 40MB of real memory.

The system is especially intersting from a software engineering point of view, since its specification and design are continually evolving as new ideas and algorithms from the natural language field are added to the system. This process has been greatly assisted by the use of a lazy functional language, sometimes in rather surprising ways.

One example of this is in the implementation of the semantic network. Its functional implementation and the need to refer to it explicitly rather than as an implicit part of the state was initially seen as a serious disadvantage. Much to our surprise, these have turned out to be very useful when dealing with multiple interpretations of text, since we can produce a different version of the semantic network for each interpretation. The functional implementation ensures that very little of the structure is duplicated and the explicit references to the semantic network ensure that there is no difficulty in deciding which version to use at any point.

Another interesting feature of our implementation is its use of lazy evaluation. Although some of the major analysis phases of the system are contained in quite separate modules, some of the later phases effectively control the amount of work done by the earlier phases. For example the parser produces a data structure which contains many possible parses. The semantic analysis will pick the first of these and if this is successful the parser will not have to do any of the work associated with producing the other parses. Alternatively, some later analysis phase may reject the output of the semantic analysis and this may have no alternative but to look for another acceptable parse, in which case the parser will then have to do more work to produce the details of other alternatives. In this way lazy evaluation has allowed us to separate the generation of data structures such as parse tree alternatives from the decision of how much of the data structure is actually required. This separation has been achieved without any need for explicit passing of control back and fourth between analysis phases.

Hydra: Haskell as a Computer Hardware Description Language

John O'Donnell, University of Glasgow

Traditional circuit designers used schematic diagrams as their computer hardware description language (CHDL). Most modern design uses the industry-standard language VHDL, and some academic research is based on formal systems like HOL and Ruby.

Hydra is an alternative approach: it uses Haskell as a CHDL, augmenting it with a library of functions and types. This approach is sensible because a practical, full-blown programming language is required to support the wide ranging requirements of circuit design, while a clean semantics, good type system and equational reasoning make it much easier to design reliable circuits. Specific benefits of Hydra include:

1. *Readable specifications.* Circuit specifications can be written using concisely using notation close to ordinary boolean logic equations. A complete CPU design requires only 2 or 3 pages, instead of the 200 or 300 required by conventional methods.

2. *Executable specifications.* A circuit specification is a function from inputs to outputs; it is executed by applying it to suitable inputs.

3. *Equational reasoning.* It is quite feasible to derive difficult circuits from simple specifications, providing both a correctness proof and an explanation of how the circuit works.

4. *Circuit patterns.* Higher order functions provide a way to express common patterns of circuits. Two families of functions (the linear combining forms and the tree combining forms) suffice for processor architecture design.

5. *Signal representations.* Several different kinds of circuit simulation are required from time to time, as well as various analyses. These can be achieved by using Haskell type classes to provide a class of Signal representations.

The constraints of working in a pure functional language cause several interesting technical problems, including:

1. *Running clock signals through combinational logic.* Although poor style, this trick was common in the 1960s, so it's still taught by many engineers. It is possible but ugly to express it with Haskell.

2. *Asynchronous circuits.* Hydra operates at a higher level of abstraction, making it awkward to deal with multiple clocks.

3. *Graph traversal for netlist generation.* Traversing circular graphs is tricky in a pure language, but there are several approaches for doing it.

High Performance Fortran: an opportunity for functional programming?

Rex Page, University of Oklahoma

Organizations that do scientific computation have invested heavily in Fortran software and training. Advocates of functional programming can take advantage of the inertia that this investment engenders by building compilers that generate efficient code for programs written with a functional subset of the ISO standard for Fortran (Fortran 90) and by writing, in Fortran 90, kernels and library components for important applications that demonstrate how to use the functional subset effectively.

This presentation discusses a subset of Fortran 90 that supports functional programming and compares it to other functional programming languages. It also explains that now is a crucial period because the High Performance Fortran Forum, a group that developed in 1993 a collection of extensions to Fortran 90 to deal with data distribution on distributed memory computers, will develop in 1994 further extensions to Fortran 90 to deal with parallel tasks. The Forum seeks proposals for these extensions and convincing demonstrations that proposed extensions address the problem of irregular parallel computation (i.e., computation outside the realm of data parallelism) adequately.

A collection of kernels written in a functional style for a variety of important applications could demonstrate that the type of hierarchical multitasking available in functional programs via parallel evaluation of operands/arguments in expressions, possibly augmented by some sort of mechanism for controlling parallelism in a way that is reasonably compatible with functional programming, provides adequate expressive power to satisfy the needs of high performance, scientific computing. Such a demonstration might dissuade the Forum from adding to HPF less disciplined multitasking facilities, such as message passing primitives. Success in this endeavor could encourage the emergence of functional style in Fortran software development for parallel computers.

Discrete Event Simulation via Functional Programming

Werner Pohlmann, TU München

Discrete event simulation is a computational technique for the quasi- empirical study of systems whose behavior may be characterized by events (state changes at irregular isolated points in continuous time); one wants to learn what happens when and ultimately get derived figures like system throughput etc.. In this field, a program in a procedural language with built-in simulation mechanism (cf Simula) traditionally is the only formal account of the model in question, and evolution of methods has focussed on userfriendliness, technical support and power

rather than model development. We want to define models abstractly, with the simulation mechanism factored out and with semantics that are easy enough to encourage and support formal reasoning in model conception, variant formation etc.. This should be useful in the initial phase of major simulation projects and especially help to develop models for the new possibilities of distributed simulation.

We use functional programming (through Haskell) in the obvious way. Model components are described by functions that process streams of what -when data and are combined into systems of equations that define the behavior of the real system under study. As an application, we have redefined the simulation-related parts of the RESQ package in this style. (RESQ is a classic software tool for queueing network and especially computer system performance studies, which actually form the context of our present research.)

Such model definitions are fine as specifications, but their use as executable programs typically requires semantic strengthening. Discrete event simulation models are concerned with points of time when something happens, but the dynamics of the real system may rely on non-occurrence of events for some interval (a lcfs server e.g. may advance a customer only after exclusion of any intervening job). Our Haskell definitions, therefore, may on execution fail to cover the complete time axis (viz will give the least fixed point instead of the maximal, which, under natural assumptions about the modelled system, uniquely exists and can be computed). This difficulty is related to the deadlock problem in distributed simulation, and we can borrow from solutions found there. The "conservative" strategy relies on additional no-event messages which exploit the lookahead capacity of some model components. This is basically simple but requires insight and careful tuning to avoid a deluge of additional data and processing. The "optimistic" strategy identifies the next due event as the earliest one in a set of tentative events deduced under the assumption of no further events. This gives a kind of relaxation algorithm for stream equations. At present we are exploring this idea. Future plans include the definition of experimental frames or testing the approach for other application areas.

Functional Programming Applied To Numerical Problems

J.A.Sharp, University of Swansea

As part of the Functional Languages Applied to Realistic Exemplars (FLARE) Project in the UK the group at Swansea has developed a computational fluid dynamics (CFD) exemplar in Haskell. A sequential version of a Taylor-Galerkin/ pressure correction algorithm has been recoded in Haskell. This algorithm is used to solve incompressible fluid flow problems described by the Navier-Stokes equations. It was originally implemented in Fortran and used extensively by the CFD research group in the Department of Computer Science at Swansea. It

involves solving large sparse systems of linear equations.

The Navier-Stokes equations are first semi-discretised in time, and then subsequently solved in space for each such interval. The spatial domain over which the solution is desired is triangulated in 2D into a finite element mesh. As a result, the fluid dynamics problems are transformed into a set of four fully-discretised matrix equations on each time step. These system matrices are large, banded and symmetric and the equations are solved by employing Jacobi iteration and Choleski direct methods.

In this project, various implementation issues in a functional environment have been investigated. They cover data structure issues, such as the implementation and performance of a generalised envelope storage scheme for Choleski decomposition, code parallelisation on a GRIP MIMD platform and the use of a Quadtree data structure for Choleski factor storage. We are currently utilising programming tools provided by the University of York as part of their contribution to the FLARE project. These tools help the development of the parallel code and further work on code optimisation.

A reasonably efficient Haskell sequential implementation has been obtained. Our application experience confirms many commonly recognised advantages of functional programming including the expressiveness and ease of maintenance. However, some problems have been encountered as well. For example, in this numerical application, lazy evaluation has not been found to be advantageous. Forced evaluation has been used in various places to make the solution of larger problems possible. Also, a time and space efficient implementation of Haskell arrays is needed. More details about the lessons learned can be found in, and in a forthcoming book summarising the work of the FLARE project.

Our current interests are in the development of parallel implementations of numerical algorithms in Haskell. Some experiences of using a quasi-parallel compiler developed at York will be reported. This tool has been found to be very useful. We demonstrate salient features of the functional approach such as the relative simplicity for parallelisation and suitability as a prototyping tool. We have also found lazy evaluation to be necessary in some code sections for efficient parallel evaluation.

As part of this work we plan to develop a system to execute a Haskell program in parallel on a network of workstations. We also aim to expand the range of numerical applications we have experience of and to this end we have been awarded a Science and Engineering Research Council (SERC) grant in conjunction with the Civil Engineering Department at Swansea to investigate "Distributed Parallel Processing for Computational Fluid Dynamics".

Towards Software Tools in Haskell

Peter Thiemann, Universität Tübingen

In order to increase the acceptance of lazy functional programming languages it is necessary to present efficient implementations for typical programming problems. We discuss the design and implementation of several software tools and a medium-scale real-life application in Haskell: an m4-like macro processor, the grep/fgrep utility, and a tool to automatically create railroad diagrams from EBNF syntax descriptions.

Declarative Real-Time Systems

Staffan Truve, Carlstedt Elektronik AB

Carlstedt Elektronik in Sweden are developing a functional language and an architecture for embedded real-time systems. The talk gives an overview of the language, concentrating on the I/O-model, and describes an architecture supporting parallel evaluation of the language.

Implementing Tools by Algebraic Specification

Pum H.R. Walters, CWI, Amsterdam

ASF2C is a compiler which translates algebraic specifications (ASF) into (ANSI) C. An ASF specification consists of a set of conditional rewrite rules, where conditions are (in)equalities of terms. The definition of a function consists of the set of rules pertaining to that function. There is no constructor discipline.

ASF is a sublanguage of ASF+SDF, which allows the definition of arbitrary (context free) syntax for terms, and supports a module concept. ASF+SDF specifications are commonly developed using the ASF+SDF meta-environment, which provides syntax-aware editors, an interpreter and other development tools.

The version of ASF2C described here is the first stable prototype. It has taken 8 man-months to develop, and it is entirely implemented in ASF. The compiler is self-compiling.

Tools made with ASF2C

• SEAL is a description language for the interface between generic editors and other computational components. SEAL was developed to describe programming environments generated with the ASF+SDF meta-environment. Size: 12000 lines of ASF+SDF code, distributed over three components.

• The pretty-printer generator produces, given an (ASF+)SDF specification, a pretty printer (implemented in ASF+SDF) for the language described by the source. Size: 4300 lines.

• ASF2C itself was of course also implemented using ASF2C. It was developed using the ASF+SDF meta environment, and was subsequently bootstrapped. Size: 3000 lines.

• fSDL is an extensible language for parameterized datastructure descriptions (which are translated into libraries with C code). A functionally complete prototype (size: 4000 lines) was satisfactory, but subsequent upscaling in an industrial project failed.

Major identified shortcomings

• The output of ASF2C is generally a large piece of C code which needs to be translated further (by a C compiler) before it can be used. This latter stage typically requires 90% of the overall time, and often requires more resources (time and memory) than available.

• Currently, the code generated by ASF2C does not support garbage collection. For small to medium sized applications this is not an issue, since calculations are finished before memory is exhausted.

• The recursive, purely functional method of specification in ASF+SDF has a very influential bad side-effect: recursive construction of a flat list implies quadratic memory consumption in the listsize.

Erlang - A Functional Programming Language for Developing Real-Time Products

Mike Williams, Erlang Systems Division, Ericsson Infocom

Software development is by far the largest part of the design costs for Ericsson's products. A single AXE-10 telephone exchange, for example, has on average software corresponding to 62 million lines of source code in C, C++, EriPascal and PLEX. Reducing the cost of software design, improving quality (fewer bugs) and cutting time to market are thus of paramount importance.

Using a programming technology which results in smaller and more understandable programs would improve the situation. A functional language is an obvious alternative.

We have designed the functional programming language 'Erlang' for this purpose. Using Erlang as the implementation language in several real-time system products has resulted in considerable reduction both in the size of programs and in the work required for software design.