Final report on the GRASP project

Simon L Peyton Jones Philip Wadler
Cordelia Hall Kevin Hammond
Will Partain
Department of Computing Science, University of Glasgow

August 5, 1993

Abstract

This is the final report on the GRASP project, carried out under SERC grants GR/F34671 and GR/F98444, at Glasgow University[1] [2].

The project supported two Principal Investigators (Simon Peyton Jones and Phil Wadler), and three Research Assistants (Kevin Hammond, Cordelia Hall and Will Partain). Four research students have worked in close association with the project.

1 Summary

The purpose of GRASP was to help get the technology of functional programming out of the lab and into the hands of practitioners, by producing robust and usable compilers and profilers for these languages, on both sequential and parallel systems.

The main achievements of the project are as follows:

- We have played a key role in the development of a common international non-strict functional language, Haskell (Hudak et al. [1992]). This standardisation effort has led directly to a considerable focussing of the international research community in these languages. More details about Haskell are given in Section 3.

- We have developed and distributed a world-class compiler for Haskell (Peyton Jones et al. [1993]). Section 4 elaborates.

- We have developed and made available a parallel implementation of Haskell, running on our GRIP multiprocessor (Section 5). (GRIP was designed and built in the Alvey-funded GRIP project, during 1985–88.) In spite of the fact that GRIP's hardware technology is now somewhat dated, GRIP is still the fastest parallel implementation of a non-strict functional language in the world, in absolute (let alone relative) terms[3].

- We have made several scientific advances in the area of language design and implementation, whose significance and usefulness transcends our particular implementations. For example: first-class unboxed values; the use of monads to accommodate input/output, state-ful computation, in-place update for arrays, and mixed-language working, all within a purely functional language; monads as a technique for structuring large programs; two separate approaches to deforestation, the removal of intermediate data structures; a new analysis to detect when updates can be avoided; a new space and time profiling technique for lazy languages; and a notation and operational semantics for graph reduction. These are discussed further in Section 6.

- We believe that detailed, quantitative measurements of the actual behaviour of "real" functional programs is now sorely needed. While this was not the main goal of GRASP — it is the subject of the newly-begun AQUA project — we made some progress in this direction (Section 7).

- We have throughout worked in active col-

[1] The original GRASP proposal included a component at Essex University, funded under a separate SERC grant number, which is the subject of a separate Final Report.

[2] This report covers two SERC grants, so it is a bit longer than usual.

[3] Zapp, another SERC-funded project, is the only real competitor, but it is restricted to divide-and-conquer parallelism only (McBurney & Sleep [1987]).

laboration with other research workers. In particular, we have enjoyed a productive partnership with the FLARE project, whose goal was to write realistic applications programs in Haskell, for both sequential and parallel platforms. FLARE is coordinated by Colin Runciman at York, with partners at British Telecom (Martlesham Heath), the University of Kent, and the University of Swansea.

There is also a committed group of four PhD students at Glasgow, who have benefited directly from the practical and intellectual support GRASP has given, and who have in turn benefited the project (Section 4.3).

- The most durable and transferable outcome of a research project is often its publications. The best technology in the world is no use to others unless its principles are published in articulate and intelligible form.

  We have been quite successful in this regard, publishing a book, 35 papers, and 17 technical reports over the life of the project (Section 8). All are available electronically by FTP, as is the source code of our compiler and associated tools.

We are well satisfied with GRASP. The project has been extremely productive, in terms of robust software, scientific advances, and published papers. It has maintained and enhanced the UK's leadership in the functional programming area, and taken significant steps towards making functional programming tools available to, and usable by, ordinary programmers.

2 Plans versus reality

This section compares the original proposal with what actually happened. (Quotations are from the proposal.)

2.1 Goals

The purpose of the GRASP infrastructure project was "to enhance the functional and logic language implementations built by GRIP and BRAVE into robust, high-performance systems".

The goals were stated as follows:

- "To develop the compilers, run-time systems and profiling tools required to support the wide use of declarative programming, on both sequential machines and the GRIP multiprocessor."

  Achieved.

- "To achieve high *absolute* performance, rather than simply demonstrating relative speedup as processors are added. Specifically, our target is that a fully-populated GRIP machine should run ten times as fast as a comparable sequential machine. By 'comparable' we mean a Sun 3 running code compiled by a good functional-language compiler."

  Exceeded. GRIP exhibits absolute speedups of up to 17 with 20 processors, relative to the same program compiled for a purely-sequential environment with equivalent processors.

- "To make these tools freely available to others outside the project who are interested in experimenting with parallel declarative programming."

  Mostly achieved. Both our compilers and GRIP have been available to others throughout the project, and we make regular releases of the Glasgow Haskell Compiler. The caveat is that, until recently, GRIP's has been much less reliable than we had hoped (Section 2.4).

- "To feed back the experience of actual use into the refinement process of the tools and systems, so that their final versions reflect the requirements of users."

  Done. This is a hard goal to measure, but we have certainly based many of our strategic decisions on feedback from our users.

- "To carry out research into the performance and resource-management issues which are of critical importance in any parallel system, in the specific context of parallel graph reduction."

  Achieved. Section 5.3 gives details.

- "To realise the potential of the existing DTI and SERC investment in GRIP hardware and software."

  You should judge!

2.2 Specific project deliverables

The original proposal gave the following list of deliverables:

- "A *compiler* for Haskell, capable of generating high-quality native machine code for a Sun workstation."

  Delivered.

- "A *programming environment* for Haskell supporting a more interactive, exploratory and incremental style of program development than a batch-oriented compiler."

  Not done. See Section 2.3.

- "A compiler for Haskell to the *GRIP parallel hardware*, producing a substantial absolute performance improvement over the sequential implementation."

  Delivered.

- "A suite of *performance-monitoring tools*, which gives the systems programmer and applications programmer feedback about the execution of the program on the GRIP hardware."'

  Delivered.

2.3 Changes of plan

We made one major change of plan, by deciding not to build an interactive Haskell system to complement our compiler. Whilst we see such a system as extremely desirable, it became apparent that it was a very large task, and carrying it out would spread our effort too thinly. Furthermore, early in the project, an excellent interactive implementation of a language very similar to Haskell (Gofer) became available. This system satisfies many of the needs for which we had intended our proposed interactive Haskell system.

2.4 Disappointments

We have had one major disappointment. From an early stage we offered our FLARE partners access to GRIP. The Haskell implementation on GRIP was based on our *prototype* Haskell compiler (see Section 4.1), and was never reliable. It seemed that whenever we tried a new functional program, the system would fail in a new way!

We invested very considerable effort in addressing this problem. In the end (late 1992), we decided to cut our losses and instead re-target the new Glasgow Haskell Compiler (Section 4.2) to GRIP. (We had repeatedly postponed doing so in the hope that "one more fix" would complete the old system.) While the re-targeting was a substantial task, it has been extremely successful, and GRIP is now highly reliable. With the benefit of hindsight we should have done this somewhat earlier, though the new compiler was not in a fit state to re-target until the second half of 1992.

3 The Haskell language

This section and those which follow give some more details of the work we have done. We begin with the Haskell language itself.

Haskell is a new, common, international, non-strict functional programming language. Its major goal was to reduce the harmful diversity of non-strict functional languages which arose in the late 80's.

We (Wadler, Hammond and Peyton Jones) played an important role in its design. The language went through three major iterations, with a Report produced at each stage. Wadler was the Editor for Version 1.0 and Peyton Jones for Version 1.2 (Hudak edited Version 1.1). Haskell's major innovation is its systematic new approach to overloading, based directly on a proposal by Wadler.

Haskell has had a very valuable focussing effect on the international research community. It has rapidly become to the non-strict community what Standard ML is to the strict community. It is also a key stepping stone on the path to serious practical applications of non-strict languages: people have become confident enough in its stability and its implementations to write substantial applications in Haskell, which was simply not the case before. In supporting this effort, we believe that GRASP has rendered an important service.

4 The Glasgow Haskell compiler

A major component of the GRASP project was to produce a robust, state-of-the-art compiler for Haskell. This goal is more easily stated than

achieved. Even leaving aside its new type system, Haskell is a rather large language, incorporating: a rich syntax; a wide variety of built-in data types including arbitrary-precision integers, rationals, and arrays; a module system; and an input/output system.

4.1 The prototype compiler

We needed a Haskell compiler at an early stage in the project, and we constructed one by making substantial extensions to an existing compiler, the Chalmers LML compiler (Hammond & Blott [1990]). This work resulted in the first implementation of Haskell in the world.

4.2 The Glasgow Haskell Compiler, `ghc`

We always knew that the prototype compiler would be unsatisfactory in a number of ways — it is a modification of an already heavily-modified compiler — so we undertook to build a completely new compiler for Haskell, written in Haskell. This is the Glasgow Haskell Compiler, `ghc`.

A substantial proportion of the total GRASP effort went into `ghc`. Haskell is a large language, so the compiler has to be large too: `ghc` is the world's largest Haskell program with upward of 30,000 lines of source code. An overview of `ghc` is given in Peyton Jones et al. [1993].

One early decision we took was to generate C as our target code (Peyton Jones [1992a]). This has paid off handsomely. We stand to achieve wide portability, and by using the Gnu C compiler, `gcc`, we can still generate excellent machine code. The main problem is that compilation time is longer than we would like, because we generate a lot of C and a lot of time is therefore spent in `gcc`. We are considering writing some "quick-and-dirty" native code generators, simply to speed up compilation.

We made "private" releases of `ghc` to our friends and partners during 1992, and the first public release (`ghc 0.10`) was in December 1992. Shortly after we began `ghc`, Lennart Augustsson at Chalmers University also started work on a Haskell compiler (based on his earlier LML compiler) called `hbc` (Augustsson [1993]). The friendly rivalry between `hbc` and `ghc` has been highly productive: both compilers have made enormous strides in the quality of the code they produce, and are still neck and neck.

The `ghc` effort has been much more important for us than simply producing a compiler as a product. It has served as the spur which has led us to develop a number of new ideas, discussed later in Section 6. It has also provided an infrastructure for the work of others, a matter to which we now turn.

4.3 The compiler as motherboard

As we began the design of `ghc` we evolved a secondary aim: that it should be built in a sufficiently modular fashion that others could use it as a "motherboard" into which to "plug" their new whizzy program analyser, optimiser, or whatever. Much research lies untested because too much effort is needed to build the scaffolding of a suitable front end and back end. Instead, a tiny front end and back end are usually built for a toy language, but the results are unconvincing because there are no credible programs written in the toy language, and the results might in any case be swamped by other optimisations which a serious compiler would perform.

In contrast, using our compiler as a framework allows a new technique to be evaluated in the context of a fully-optimising compilation path, and using a large suite of serious test programs (Section 7.2).

We have been quite successful in fulfilling this aim. We know of the following definite "users" of the the compiler as motherboard:

- Jim Mattson, a PhD student at the University of South California at San Diego (UCSD), ported our entire parallel implementation (the prototype version) to a BBN Butterfly machine, and used it as a basis for his PhD work on speculative computation (Mattson [1993]). Whilst this was a use of the prototype compiler, not `ghc`, it is an example of the same process in action. (Subsequent to this work, Jim has joined the AQUA project at Glasgow.)
- Jon Hill, a PhD student at Queen Mary and Westfield College, is using `ghc` as the basis for a data-parallel extension of Haskell, targeted for the DAP (Hill [1993]).
- Simon Marlow, a PhD student at Glasgow,

has implemented a new update analyser, which figures out when thunks can safely be evaluated without update (Launchbury et al. [1993]; Marlow [1993]).

- Andy Gill, another PhD student at Glasgow, has implemented a new technique for removing intermediate lists in non-strict functional programs (Gill & Jones [1993]; Gill, Launchbury & Peyton Jones [1993]).

- Andre Santos, a PhD student at Glasgow, is using the compiler as a basis for his work on automatic program transformation (Santos & Peyton Jones [1993]; Santos [1993]).

- Patrick Sansom, a PhD student at Glasgow, has used the compiler as a basis for his generational garbage collector, and made detailed performance measurements of its behaviour on large programs (Sansom [1991]; Sansom & Peyton Jones [1993b]).

- Patrick Sansom has also implemented a new time and space profiling technique, based on so-called *cost centres*, and used it to profile and tune large programs (Sansom & Peyton Jones [1993a]).

- All the above are "solid"; that is, we have had modified, working source code back from them. At a "softer" level, we have had (at least) the following expressions of strong interest:
  - two groups have expressed definite interest in using `ghc` as a base for a hardware description language (both USA);
  - another intends to develop a real-time generational garbage collector (Australia);
  - another is implementing "constructor classes", an extension of the Haskell type system (Australia);
  - another intends to integrate his sophisticated strictness analyser into `ghc` (Manchester);
  - and about ten groups are working on porting the compiler to other platforms.

4.4 Literate programming

Knuth introduced the idea of "literate programming", in which a program is written as a single document, which can either be fed into a compiler, or typeset to make a human-readable text, with some pre-processing being necessary in either case (Knuth [1984]). Based on this idea, we have developed a simple literate programming system for Haskell. In most respects it is simpler than Knuth's system, but we have extended the idea in one way: the program can be read in a *third* way, as a simple hypertext using the Gnu `info` system.

The entire compiler is written using this system.

5 The GRIP multiprocessor

It has long been claimed that functional languages are particularly suitable for parallel evaluation. The absence of side effects means that distinct sub-expressions of the program can be evaluated concurrently; no new language constructs (threads, semaphores, critical regions and the like) are needed; no new semantic complexities are introduced; and the program remains deterministic, in stark contrast to the unpredictable behaviour of parallel programs in other languages.

Despite this promise, and many prototypes, there are very few "real" parallel implementations available. By "real" we mean implementations which give an *absolute* speedup over the same (functional) program compiled by a state-of-the-art compiler for a sequential processor made from comparable technology. Indeed, the only two such implementations known to us are the $\langle \nu, G \rangle$ machine (Augustsson & Johnsson [1989]), and our own machine, GRIP. Furthermore, the former is now effectively defunct. (Zapp, another SERC-funded project, is another parallel implementation delivering good performance, but it is restricted to divide-and-conquer parallelism only (McBurney & Sleep [1987]). Other work is in progress, of which the most promising is that of the Nijmegen group.)

Why are "real" implementations so few? It is, in our view, mainly because building such a system is a major undertaking. Functional languages shift much of the burden of resource management from the programmer to the system. This shift is even more pronounced in the case of parallel systems, where the system is responsible for the scheduling and location of threads, for the placement of data, and for all communication

and synchronisation. As a direct result of all this, the language implementation on the parallel system becomes quite complex.

5.1 The hardware system

The basis for our work in GRASP was the GRIP multiprocessor itself, built under Alvey support in the GRIP project. Despite its old technology (GRIP uses M68020 processors), GRIP's absolute performance still outstrips that of a (single) Sparc 10, for suitably-parallel programs at least.

From an architectural point of view, the most interesting part of GRIP is the use of intelligent memories (IMUs). The idea is *to raise the abstraction provided by memory* from the usual vector-of-words, supporting two operations (read and write), to a garbage collected heap supporting many operations (allocate, fetch and lock, update, garbage collect, and so on). To this end, the IMUs consist of 5 Mbytes of RAM, very closely coupled to a microprogrammed data engine. GRIP's hardware architecture, and that of the IMUs in particular, is described in Peyton Jones et al. [1987].

The ability to add new microcoded memory operations has proved extremely valuable. For example, we recently added microcode which allows the IMUs to circulate among themselves a measure of the overall system load. Since this is done entirely in microcode with no PE intervention, it happens extremely fast, which in turn directly boosts performance of parallel Haskell programs.

There is one design decision which turns out to have been badly mis-judged: there is too little memory on each PE (1 Mbyte). When GRIP was designed this seemed quite a lot, but it is now the factor which painfully limits how big a Haskell program can be run on GRIP. There is no problem with the GRIP architecture itself, but in cold reality there is no prospect of being able to increase the memory on each processor. What a classic error!

The hardware itself has been reassuringly reliable. Some individual boards have developed faults, but they can easily be taken out of service. Once this is done, the remaining system works faultlessly. We have been hampered in fixing the faulty boards (in theory we have enough hardware for 32 processors) by the fact that GRIP's hardware designer left at the end of the GRIP project. The only person who understands the hardware in detail is Simon Peyton Jones, who has too many other things to do!

We have continued to use the GRIP Lightweight Operating System (GLOS) developed in the GRIP project, and much associated system software.

5.2 Haskell on GRIP

We have built an implementation of compiled parallel graph reduction to run on GRIP. The design is described in Peyton Jones, Clack & Salkild [1989].

The most significant aspect of the design is the memory hierarchy. The Intelligent Memories (IMUs) support a *global heap*, which is cached in the *local heap* on each PE. There are no pointers into these local heaps from outside a PE, so they can be garbage-collected independently; this means that locally-generated "litter" is recovered very cheaply. Only when the entire global heap fills up is global garbage collection initiated, which requires the cooperation of all the PEs and all the IMUs. The IMUs do much of the global garbage collection autonomously, in microcode.

A runtime system, replicated on each PE, manages the scheduling of parallel threads, storage management, and the movement of data between global and local memory. The first version of the runtime system was written rather too rapidly, at the very end of the GRIP project. It never worked satisfactorily, despite the investment of substantial effort. Late in 1992, we cut finally our losses and rewrote the runtime system in C, integrating it with the new Glasgow Haskell compiler. The resulting system now works very reliably.

5.3 Experience and results

Despite these difficulties, we have succeeded in running a variety of experiments on GRIP, leading to a succession of papers (Akerholt et al. [1991]; Akerholt et al. [1993]; Hammond & Peyton Jones [1992]). Any parallel functional language implementation uses a whole raft of strategies and heuristics to manage the resources of the system. These papers offer preliminary exploration of the effect of some of

these strategies in the context of a real system.

An important thread in this work has been measurements of a parallel database system based on ideas explored in Trinder's thesis (Trinder [1989]). The key idea is to represent the database by a tree, which allows *inter-transaction* as well as *intra-transaction* parallelism to be exploited. The results of this work have been sufficiently encouraging that they have led to a recently-funded SERC project — PARADE — to build a more substantial and realistic functional database system on ICL's EDS hardware platform.

Recent experiments on GRIP with optimised list constructors called 'hydras' have yielded a factor of 4 speedup on a substantial program. These constructors allow several list cells to be represented as one, reducing GRIP communication overhead (Hall [1993]). Thus it seems likely that the years of theoretical work on strictness analysis over data structures will pay off in practice.

5.4 The future

We are by no means finished with GRIP. We will continue to make GRIP available to our partners, and extend the range of parallel applications which it runs. We intend to make further detailed measurements of the effect of various resource-management strategies. We may explore more substantial extensions, such as adding speculative evaluation (Mattson [1993]).

The major tactical decision is when to start investing heavily in some other, more widely available, parallel platform. But that is beyond the scope of this report.

6 Scientific advances

Computer Science is both a *scientific* and an *engineering* discipline. As a scientific discipline, it seeks to establish generic principles and theories that can be used to explain or underpin a variety of particular applications. As an engineering discipline, it constructs substantial artefacts of software and hardware, sees where they fail and where they work, and develops new theory to underpin areas that are inadequately supported. (Milner [1991] eloquently argues for this dual approach in Computer Science.)

One of the delights of functional programming is that it offers an unusually close interplay between these two aspects (Peyton Jones [1992b]). Theory often has immediate practical application, and practice often leads directly to new demands on theory. We have seen many examples of this interplay, and our work in GRASP has driven us to make a number of scientific advances, which have a much wider applicability than our project alone. Much of this work has been done in collaboration with our PhD students: Andre Santos, Andy Gill, Patrick Sansom, and Simon Marlow.

6.1 The STG language

We have developed a new notation and operational semantics for graph reduction, called the Shared Term-Graph language, or *STG language*[4] (Peyton Jones [1992a]). The interesting thing about the STG language is that it is a purely functional language with both a conventional *denotational* semantics, and an *operational* semantics, expressed as a state-transition system, akin to that conventionally used for abstact machines. It therefore forms an ideal stepping stone in the compilation path.

6.2 Unboxed data types

In order to expose to the compiler's transformation system the boxing and unboxing of data values required by arithmetic operations, we developed a notation and theory for *unboxed data types* (Peyton Jones & Launchbury [1991]). This notation significantly extends the expressiveness of both the source language and the compiler's intermediate language, exposing to the compiler a number of optimisations which previously lay buried in dark corners of the code generator.

6.3 Hydras

We have extended the unboxing approach to include an optimised representation of lists (Hall [1993]). Each constructor contains several list heads and a single tail, 'unboxing' the other tails and coalescing them into one structure. These have been shown to cut execution time

[4] The "STG" originally came from the abstract machine used to implement the STG language, the Spineless Tagless G-machine. In fact the language is quite independent of the implementation technology, so a separate name is appropriate even if, for historical reasons, they share a single acronym!

in half on a substantial program. We have also developed a general framework for optimising abstract data types automatically. This new analysis and transformation was inspired by the original design of the typechecker, which translates into the second order polymorphic lambda calculus.

6.4 Monads for state and input/output

Based directly on work by Moggi and Wadler (Moggi [1989]; Wadler [1992a]; Wadler [1992b]), we have demonstrated and implemented a new model for *input/ouput in purely functional languages*, based on monads. Not only has this approach made I/O-performing programs easier to write, but has also led us to semantically clean ways of incoprorating mutable arrays, state-ful computations and mixed-language working into Haskell (King & Launchbury [1993]; Launchbury [1993]; Peyton Jones & Wadler [1993]). This is a very exciting new area.

6.5 Profiling

Everyone knows that it is often possible to make substantial improvements in the performance of a program by changing only a small proportion of its code. The trick, of course, is to know just *which* part of the code to pay attention to. This question can be particularly difficult to answer in a lazy functional program, because the demand-driven evaluation mechanism leads to a rather non-intutive execution order. In response to this need, we ave developed a new *space and time profiling technique*, based on so-called "cost centres", which allows accurate profiling of the space and time consumed by different parts of the program, despite the fine interleaving imposed by lazy evaluation (Sansom & Peyton Jones [1993a]).

Runciman and Wakeling at York have also been working on profiling systems for Haskell, and have produced an excellent space-profiling tool which gives similar information to ours (Runciman & Wakeling [1993]) — indeed, we use their text-to-graph mangler as part of our system. The unique feature of our work is that it allows *time* consumption to be profiled as well; theirs cannot easily be extended to do this.

6.6 Deforestation

Functional programs are often constructed by combining smaller programs, using an intermediate list to communicate between the pieces. Unfortunately, these intermediate lists have a run-time cost, both in space and time.

We have invented and implemented a new *deforestation technique* for removing many of these intermediate lists in non-strict functional programs (Gill, Launchbury & Peyton Jones [1993]). It is particularly effective in removing the intermediate lists which otherwise are generated by Haskell's array comprehensions.

We have also extended Wadler's earlier work on deforestation (Wadler [1990]) to deal with higher-order functions, though the implementation is not yet complete (Marlow & Wadler [1993]).

6.7 Update avoidance

Lazy evaluation creates many *thunks*, or as-yet-unevaluated heap objects, which are subsequently updated with their final value. However, our measurements indicate that over 70% of thunks are only used once, and hence need not be updated (a considerable cost saving). We have developed a new analysis, update avoidance analysis, which identifies a substantial fraction of these single-use thunks, and implemented it in `ghc`. It works rather well, allowing the compiler to omit 10-60% of all updates (the exact figure varies widely from program to program) (Marlow [1993]).

7 Measurements

There is a simple idea which has changed the world of computer architecture: to obtain high performance, one must make quantitative measurements of the behaviour of "real" programs, and make sure that the most common operations are performed at blinding speed — even if less common operations go a bit slower as a result. This common-sense insight is *the* breakthrough which launched the RISC revolution in computer architecture.

We believe that it is essential to make these kinds of detailed, quantitative measurements of the behaviour of functional programs. The AQUA project, which has just begun and is

funded by SERC, is precisely aimed at this goal. In GRASP we carried out some initial work in this direction, which is outlined here.

7.1 Generational garbage collection

The 1980's have seen the successful development of *generational* garbage collection techniques (Lieberman & Hewitt [1983]; Moon [1984]; Ungar [1984]), which is now well established in the symbolic processing community. Curiously, though, there have been few attempts to examine the suitability of generational garbage collection for implementations of non-strict functional languages, such as Haskell. This may be due to the observation that common implementation techniques for non-strict functional languages perform many write operations to already-existing objects. This is precisely the operation that generational garbage collectors make expensive.

We have implemented a generational garbage collection system, and made extensive measurements of its performance when running substantial Haskell programs. Contrary to popular belief, we found that lazy functional language implementations are rather good subjects for generational garbage collection. Objects so young when they are updated that almost all updates occur in the new generation, thus avoiding most of the worrisome overhead. Fuller details can be found in Sansom & Peyton Jones [1993b].

7.2 The `nofib` benchmark suite

The lack of a standard language has long hindered the development of a serious benchmark suite for lazy functional languages. There are a number of obvious reasons why such a suite is desirable:

- It encourages implementors to improve aspects of their compiler that give benefits to "real" applications, rather than merely improving the performance of "toy" programs.
- It give a concrete basis for comparing different implementations.

As part of our work we have developed the `nofib` benchmark suite (Partain [1993]), a collection of application programs written by people other than ourselves, mostly with a view to getting a particular task done. The range of applications is wide, including, for example, a theorem prover, a particle-in-cell simulation, a solid geometry application and a strictness analyser. Their size ranges from a few hundred lines of Haskell to several thousand.

8 Dissemination of results

A list of project publications is attached to this report, covering the years 1990-1993. There are a total of one book, 35 published papers, and 17 technical reports, written by members of our team or PhD students working on GRASP technology.

We have been represented at the following conferences and workshops, usually presenting one or more papers:

- Principles of Programming Languages (POPL); annual.
- Functional Programming and Computer Architecture (FPCA); bi-annual (1991 and 1993).
- The JFIT conference; 1993.
- Parallel Architectures and Langauges Europe (PARLE); bi-annual (1991 and 1993).
- The "Nijmegen" workshop on parallel implementations of functional languages; annual.
- The Glasgow Functional Programming Group Workshop. This annual event is focussed mainly on work going on at Glasgow, but about 25% of the participants are invited visitors from other universities and from industry. The proceedings are refereed, and are published by Springer Verlag.

The compiler itself, and its associated tools, is now quite widely disseminated, being freely available by FTP. It is hard to know exactly *how* widely, because people can grab it by FTP without telling us. We certainly know of tens of sites which are using it. The compiler will shortly also be distributed on the Free Software Foundation's "experimental tape".

9 Training

All three Research Assistants already had doctorates when they began work on the project. Kevin Hammond has since won a 3-year Fellowship from the Royal Society of Edinburgh; Cordelia Hall is now a lecturer at Glasgow; and Will Partain is employed on the AQUA project.

10 Collaboration and exploitation

We have enjoyed close collaboration with our FLARE partners, especially Colin Runciman's group at York.

ICL have continued to act as "uncle". In practical terms, this was manifested mainly by Phil Broughton's extremely helpful probing at our 6-monthly Steering Group meetings. This contact has led to ICL's involvement in the upcoming PARADE project, which builds directly on GRASP's work (see Section 5.3).

There are many potential users who feel they cannot rely on an unsupported university "product", such as `ghc`. Our ultimate goal is therefore for a company to turn our Haskell system into a commercial product. We have made contact with Harlequin Ltd, with this in mind; discussion is on-going.

References

G Akerholt, K Hammond, SL Peyton Jones & P Trinder [June 1991], "A parallel functional database for GRIP," in *Proc Workshop on the Parallel Implementation of Functional Languages, Southampton*, H Glaser & P Hartel, eds., 7–29.

G Akerholt, K Hammond, P Trinder & SL Peyton Jones [June 1993], "Processing transactions on GRIP, a parallel graph reducer," in *Proc Parallel Architectures and Languages Europe (PARLE), Munich*, 634–647.

L Augustsson [June 1993], "Implementing Haskell overloading," in *Proc Functional Programming Languages and Computer Architecture, Copenhagen*, ACM.

L Augustsson & T Johnsson [Sept 1989], "Parallel graph reduction with the $< \nu, G >$-machine," in *Proc IFIP Conference on Functional Programming Languages and Computer Architecture, London*, ACM.

A Gill & SL Peyton Jones [July 1993], "Building on foldr," in *Glasgow Functional Programming Workshop, Ayr*.

A Gill, J Launchbury & SL Peyton Jones [June 1993], "A short cut to deforestation," in *Proc Functional Programming Languages and Computer Architecture, Copenhagen*, ACM, 223–232.

CV Hall [July 1993], "A framework for optimising abstract data types," in *Glasgow Functional Programming Workshop, Ayr*.

K Hammond & S Blott [1990], "Implementing Haskell type classes," in *Functional Programming, Glasgow 1989*, K Davis & RJM Hughes, eds., Workshops in Computing, Springer Verlag, 265–286.

K Hammond & SL Peyton Jones [Sept 1992], "Profiling scheduling strategies on the GRIP parallel reducer," in *Proc 1992 Workshop on Parallel Implementations of Functional Languages, Aachen*, Kuchen, ed..

JMD Hill [July 1993], "The AIM is laziness in a data-parallel language," in *Glasgow Functional Programming Workshop, Ayr*.

P Hudak, SL Peyton Jones, PL Wadler, Arvind, B Boutel, J Fairbairn, J Fasel, M Guzman, K Hammond, J Hughes, T Johnsson, R Kieburtz, RS Nikhil, W Partain & J Peterson [May 1992], "Report on the functional programming language Haskell, Version 1.2," *SIGPLAN Notices* 27.

D King & J Launchbury [July 1993], "Functional graph algorithms with depth-first search," in *Glasgow Functional Programming Workshop, Ayr*.

Donald Knuth [1984], "Literate programming," *Computer Journal* 27, 97–111.

J Launchbury [June 1993], "Lazy imperative programming," in *Proc ACM Sigplan Workshop on State in Programming Languages, Copenhagen (available as YALEU/DCS/RR-968, Yale University)*, pp46–56.

J Launchbury, A Gill, RJM Hughes, S Marlow, SL Peyton Jones & PL Wadler [1993], "Avoiding Unnecessary Updates," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Workshops in Computing, Springer Verlag.

H Lieberman & C Hewitt [June 1983], "A real-time garbage collector based on the lifetimes of objects," *CACM* 26, 419–429.

S Marlow [July 1993], "Update avoidance analysis using abstract interpretation," in *Glasgow Functional Programming Workshop, Ayr*.

S Marlow & PL Wadler [1993], "Deforestation for higher-order functions," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Workshops in Computing, Springer Verlag, 154–165.

J Mattson [Feb 1993], "An effective speculative evaluation technique for parallel supercombinator graph reduction," PhD thesis, TR CS93-282, Department of Computer Science and Engineering, University of California, San Diego.

D McBurney & MR Sleep [1987], "Transputer-based experiments with the Zapp architecture," in *Proc PARLE Conference I*, LNCS 258, Springer Verlag, 247–259.

R Milner [June 1991], "Computer Science: The Core IT Research Discipline," Lab for the Foundations of Computer Science, Edinburgh.

E Moggi [June 1989], "Computational lambda calculus and monads," in *Logic in Computer Science, California*, IEEE.

D Moon [Aug 1984], "Garbage collection in a large Lisp system," in *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, Texas, 235–246.

WD Partain [1993], "The nofib Benchmark Suite of Haskell Programs," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Workshops in Computing, Springer Verlag, 195–202.

SL Peyton Jones [Apr 1992a], "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine," *Journal of Functional Programming* 2, 127–202.

SL Peyton Jones [Autumn 1992b], "UK research in functional programming," *SERC Bulletin* 4.

SL Peyton Jones, C Clack & J Salkild [June 1989], "High-performance parallel graph reduction," in *Proc Parallel Architectures and Languages Europe (PARLE)*, E Odijk, M Rem & J-C Syre, eds., LNCS 365, Springer Verlag, 193–206.

SL Peyton Jones, Chris Clack, Jon Salkild & Mark Hardie. [Sept 1987], "GRIP - a high-performance architecture for parallel graph reduction," in *Proc IFIP conference on Functional Programming Languages and Computer Architecture, Portland*, G Kahn, ed., Springer Verlag LNCS 274, 98–112.

SL Peyton Jones, CV Hall, K Hammond, WD Partain & PL Wadler [March 1993], "The Glasgow Haskell compiler: a technical overview," in *Proc Joint Framework for Information Technology (JFIT) Conference, Keele*.

SL Peyton Jones & J Launchbury [Sept 1991], "Unboxed values as first class citizens," in *Functional Programming Languages and Computer Architecture, Boston*, Hughes, ed., LNCS 523, Springer Verlag, 636–666.

SL Peyton Jones & PL Wadler [Jan 1993], "Imperative functional programming," in *20th ACM Symposium on Principles of Programming Languages, Charleston*, ACM, 71–84.

C Runciman & D Wakeling [1993], "Heap profiling a lazy functional compiler," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Workshops in Computing, Springer Verlag, 203–214.

P Sansom [Aug 1991], "Combining copying and compacting garbage collection," in *Proc Fourth Annual Glasgow Workshop on Functional Programming*, Springer Verlag Workshops in Computer Science.

P Sansom & SL Peyton Jones [1993a], "Profiling lazy functional programs," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Workshops in Computing, Springer Verlag, 227–239.

PM Sansom & SL Peyton Jones [June 1993b], "Generational garbage collection for Haskell," in *Proc Functional Programming Languages and Computer Architecture, Copenhagen*, ACM, 106–116.

A Santos & SL Peyton Jones [July 1993], "Tuning a compiler's allocation policy," in *Glasgow Functional Programming Workshop, Ayr*.

Andre Santos [1993], "On program transformation in the Glasgow Haskell Compiler," in *Functional Programming, Glasgow 1992*, J Launchbury & PM Sansom, eds., Workshops in Computing, Springer Verlag, 240–251.

PW Trinder [Dec 1989], "A functional database," DPhil Thesis, Oxford University.

D Ungar [April 1984], "Generation scavenging: A non-disruptive high performance storage management reclamation algorithm," in *ACM SIGPLAN Software Engineering Symposium on Practical Software Development Evironments*, Pittsburgh, Pennsylvania, 157–167.

PL Wadler [1990], "Deforestation: transforming programs to eliminate trees," *Theoretical Computer Science* 73, 231–248.

PL Wadler [1992a], "Comprehending monads," *Mathematical Structures in Computer Science* 2, 461–493.

PL Wadler [Jan 1992b], "The essence of functional programming," in *Proc Principles of Programming Languages*, ACM.