Haskell Communities and Activities Report

`http://www.haskell.org/communities/`

– first edition –

November 8, 2001

Claus Reinke (editor), University of Kent at Canterbury, UK
Manuel Chakravarty, University of New South Wales, Australia
Olaf Chitil, University of York, UK
Andy Gill, Galois Connections Inc., USA
Graham Hutton, University of Nottingham, UK
Johan Jeuring, Utrecht University, The Netherlands
Björn Lisper, Mälardalen University, Sweden
Rita Loogen and Steffen Priebe, University of Marburg, Germany
Jan-Willem Maessen, Massachusetts Institute of Technology, USA
Simon Marlow, Microsoft Research Cambridge, UK
Johan Nordlander, Oregon Graduate Institute, USA
Sven Panne, Beta Research, Germany
John Peterson, Yale University, USA
Simon Peyton Jones, Microsoft Research Cambridge, UK
Robert Pointon, Heriot-Watt University, UK
Chris Reade, Kingston University, UK
Martin Sulzmann, University of Melbourne, Australia
Phil Trinder, Heriot Watt University, UK
Malcolm Wallace, University of York, UK

Preface

Haskell has come a long way since the September of 1987, when a meeting at a functional programming conference decided that more widespread use of the class of non-strict purely functional languages was being hampered by the lack of a common language. It is a credit to everyone involved in the development of Haskell that the language has achieved many of the goals set out for it.

A problem with this success, however, is that trying to keep up to date with what is going on within the Haskell community as a whole is becoming more and more difficult while the breadth of interests in that community keeps expanding. Sub-communities have formed, often with their own mailing lists, to work on specific issues or just to provide foci for specialist discussion. While this is necessary to get some work done, both specialists and the "just a Haskell user" find it increasingly difficult to keep track of what is going on behind the scenes of the main list.

Few people can afford spending their days reading through all of the dozens of Haskell-related mailing lists (this is not a joke: you'll find about a dozen "real" lists plus a similar number of cvs-watchlists hosted at haskell.org alone, and that isn't counting the lists hosted elsewhere or the communities that use small meetings to coordinate their work), and a lot of information is usually passed on "behind the scenes", during breaks at conferences and workshops, at working group meetings, in private emails.

This first edition of the Haskell Communities and Activities Report is the first result of an attempt to try and organise some way of getting summaries of what the various (sub-)communities are working on and posting the results back to the main Haskell list as well as on haskell.org.

The idea is as follows: twice a year, a call goes out to the main Haskell mailing list, asking all Haskellers to contribute brief summaries of their area of work, be it language design, implementation, type system extensions, standardisation of GUI APIs, applications of Haskell, or whatever. The summaries introduce the area of work, the major achievements over the previous six months, the current hot topics, and the plans for the next six months. They also provide links to further information.

So, every six months, all Haskellers should have a bird's-eye view of the Haskell community as a whole, and pointers to more in-depth information.

Not only should this help everyone to keep informed, it will also help the Haskell communities to stay in touch as they delve ever deeper into their own specialities. Once you know what's cooking, and where, it is easier to decide which communities to join, and to contribute to the areas of work you are interested in.

To the specialist communities, these bi-annual reports offer an occasion for distributing discussion documents to the main community, asking for feedback and for contributions. We have several interesting developments of this kind in the present report.

Last, but not least, the Communities Reports should also provide a forum for the creation of new communities. The best place to start looking for collaborators on completely new projects is probably the haskell mailing list, backed up by by a note in the job adverts section at haskell.org, and potentially followed by the creation of a dedicated mailing list. But topics evolve, and seeing reports from all areas side by side might suggest useful re-arrangements of existing boundaries, or highlight problems (see the comments on the situation of individual Haskellers around the world in the final chapter).

While compiling this report, the volunteers contributing the summaries did most of the work – thanks to all of them! Keeping to the spirit of lazy evaluation, however, many summaries would not be delivered unless inspected, so one of my main roles was that of providing a strict evaluation context (well, hyper-strict, I guess, unless you are happy with the weak head normal form represented by the table of contents;-). In spite of the wealth of topics covered, this first edition does not cover all current work on or with Haskell, and I hope that more Haskellers will contribute summaries of their favourite Haskell topics in the future.

I certainly learned one or two new things from this project, and I hope you'll find it an interesting read. The real question, however, is what you are going to do with this information: most of the communities that report here are looking for contributions of one kind or another (may it be bug fixes, bug reports, co-workers, discussion partners, ... or just generally helpful user feedback).

Claus Reinke,
University of Kent at Canterbury, UK

Contents

Chapter 1

General

1.1 Haskell Central - WWW and mailing lists

Haskell's central information resource is
`http://www.haskell.org`
It has the language and standard library definitions, links to Haskell implementations, libraries, tools, books, tutorials, people's home pages, communities, projects, news, a wiki, question&answers, applications, educational material, job adverts, Haskell humour, and even merchandise.
haskell.org also hosts most of the Haskell-related mailing lists and CVS repositories (15 mailing lists at a recent count, plus about another dozen of CVS-related lists). While the overall structure of the web site has been relatively stable for some time now, the maintainers John Peterson and Olaf Chitil are aiming to keep the contents in each part up to date.
Most Haskell-related information is reachable from haskell.org, and anything that isn't, should be. Do not just wait for John or Olaf to pick URLs and infos from lengthy messages in long-running threads on the Haskell lists: *send new or updated entries (category + link + short description) directly to John or Olaf.*
Perhaps we can all take the release of this first Communities Report as an occasion for going through the information on haskell.org relating to our own interests and sending in updates, where appropriate? As its says on haskell.org: "This web site is a service to the Haskell community. The site is maintained by John Peterson and Olaf Chitil. Suggestions, comments and new contributions are always welcome. If you wish to add your project, compiler, paper, class, or anything else to this site please contact us."

1.2 Revised Haskell 98 Report

It has been a long and difficult revision, with a lot of surprising "features" being discovered and removed through a series of draft documents and intensive discussions on the haskell list. Simon Peyton Jones had taken on the job:

- To correct errors or inconsistencies.
- To add clarifying remarks where the Report is ambiguous or obscure.
- With extreme reluctance, to make minor changes to the language or its libraries.

The results can be found at:
`http://research.microsoft.com/~simonpj/haskell98-revised`
"I have posted a draft version of both Reports approximately monthly since April. Now I am posting what I hope are final versions, but I want to give one last chance for you to improve my wording. I do not want to do anything new; but I am prepared to fix any errors in the changes I have made. I urgently solicit feedback on these drafts, before the end of November 2001."

1.3 Formal basis and Meta-programming support

The effective lack of a formal semantics for the whole of Haskell (as opposed to academically interesting fragments) has been a constant source of embarrassment (functional languages: solid theoretical basis, effective reasoning about programs, ...; Haskell: ???, ahem, oops).
Similarly, anyone wanting to do meta-programming on Haskell source code, e.g., to prototype language extensions, tended to write their own fragmentary Haskell frontend on top of the few existing parsers for full Haskell.
On both these topics, there seems to have been some progress recently:

- Karl-Filip Faxen, KTH Stockholm, has been working on a formalisation of Haskell's static semantics: "My static semantics is finished and the final version delivered to Graham Hutton for pulication in the JFP special issue on Haskell *[cf. section 1.4 (ed)].*

  There are a few omissions and one deviation (can't use qualified names to refer to top level bindings in the same module, but these top level bindings shadow imported bindings, so it is easy to translate a program from Report-Haskell to KF-Haskell). The omissions are strictness flags in datatypes, the newtype construct (which is indistinguishable from ordinary algebraic data types from a typing point of view) and deriving clauses (which would probably have needed another ten pages or so to specify, most of which would be conncerned with the dynamic semantics)."

- The functional programming group at York has been working on various operational semantics for Haskell to enable reasoning about space behaviour (Adam Bakewell) and about computation traces (Olaf Chitil).

  `http://www.cs.york.ac.uk/fp/`

- Andrew Tolmach, Portland State University, has been working on a formal specification of GHC's version of core Haskell, the subset into which full Haskell is translated for definition, compilation and optimisation.

  Old: " "Core" is an intermediate language used internally by the GHC compiler. It does resemble a reduced Haskell (but with explicit higher-order polymorphic types) and GHC translates full Haskell 98 into it. Currently Core has no rigorously defined external representation, although by setting certain compiler flags, one can get a (rather ad-hoc) textual representation to be printed at various points in the compilation process. (This is usually done to help debug the compiler)."

  New: "The newly released GHC 5.02 supports an initial version of a facility for dumping GHC's intermediate code (called Core) into a file for use by other tools. The Core format has been given a formal syntax and semantics (the latter in the form of a definitional interpreter). Details are in

  `http://www.haskell.org/ghc/docs/papers/core.ps.gz`

  At present, Core can only be dumped (using the -fext-core flag to ghc); ultimately, we hope to be able to load it as well, so that the output of other tools can be fed back into GHC prior to code generation. Feedback on this facility (to glasgow-haskell-users and/or apt@cs.pdx.edu) would be very welcome."

- The Programatica project at OGI, Oregon, has been working on an implementation of Haskell's static semantics in Haskell; this has already led to some clarifications of ambiguities in the ongoing revisions of the Haskell 98 language report.

  Thomas Hallgren: "The intended use is within the Programatica project [1], where we want to be able work with extended versions of Haskell. The parser (and the abstract synax, I presume) is based on a Happy parser by Simon Marlow and Sven Panne, and I guess it has something in common with the parser used in GHC. The abstract syntax has then been refactored to separate structure and recursion, to make most of the types and related functions reusable without change in extended versions of the language. Tim Sheard talked about his unification algorithm based on the same ideas at ICFP [2].

  At the moment, the parser seems to be in descent shape, although it does not yet handle infix operators in 100% accordance with Haskell 98, since that also requires an implementation of the module system...

  The implementation of the modules system is close to completion, so we will have something that can parse Haskell 98 correctly within a couple of weeks, I guess.

  There is some code for static analysis and type checking, but it is not in good enough shape to be viewed by external eyes yet...

  Other than that, I refrain from saying anything definite. My guess is that things will get done when they seem to be needed to make progress in the project. After all, we are lazy functional programmers :-)"

  [1]`http://www.cse.ogi.edu/PacSoft/projects/programatica/default.htm`

  [2]`http://cristal.inria.fr/ICFP2001/Abstracts/13.html`

- Bernard James Pope (`http://www.cs.mu.oz.au/~bjpop/`), University of Melbourne: "A few of us at Melbourne have been slowly creating a front end to Mark Jones' Typing Haskell in Haskell. It is getting close to being useable. It is intended as a stand-alone type inference/checking tool that can give detailed information about static aspects of the program. It will understand modules.

  Given Mark's approval I would like to make it available to the Haskell community. I think seperating this element of the language from the compilers is a useful thing to do, especially for program transformations that need types. It uses the Hsparser library for parsing. As for a standard AST and interface, that would be lovely but I think it is as much a language issue as a library issue."

- Michael Sperber (`http://www-pu.informatik.uni-tuebingen.de/users/sperber/`), University of Tübingen: "Note that we also have most of a frontend based on thih. It's actually a translator between hsparser's output and thih's input. It's not very well debugged or tested, but mostly complete, albeit sans the modules stuff."

The three Haskell in Haskell frontends developed independently, using similar starting points. Now that they know about each other, it would seem to make sense for the groups to join forces, in order to make best use of sparse resources. Bernie Pope has already indicated that he would be willing to contribute to a joint effort, the other groups have yet to comment. We'll see how the story continues, but in any case, the foundations for Haskell meta-programming have been improved considerably by these projects.

1.4 JFP special issue on Haskell

The Journal of Functional Programming will be running a special issue on Haskell (submission, refereeing and editing done; expected publication sometime in 2002). Thanks to Graham Hutton, guest editor for that special issue, we have the titles, authors, and abstracts for the six papers that will appear in it, and it looks to be a very interesting special issue. CFP: `http://www.cs.nott.ac.uk/~gmh/jfp.html`

A Static Semantics for Haskell *Karl-Filip Faxen*

This paper gives a static semantics for Haskell 98, a non-strict purely functional programming language. The semantics formally specifies nearly all the details of the Haskell 98 type system, including the resolution of overloading, kind inference (including defaulting) and polymorphic recursion, the only major omission being a proper treatment of ambiguous overloading and its resolution.

Overloading is translated into explicit dictionary passing, as in all current implementations of Haskell. The target language of this translation is a variant of the Girard-Reynolds polymorphic lambda calculus featuring higher order polymorphism and explicit type abstraction and application in the term language. Translated programs can thus still be type checked, although the implicit version of this system is impredicative.

A surprising result of this formalization effort is that the monomorphism restriction, when rendered in a system of inference rules, compromises the principal type property.

Developing a High-Performance Web Server in Concurrent Haskell *Simon Marlow*

Server applications, and in particular network-based server applications, place a unique combination of demands on a programming language: lightweight concurrency, high I/O throughput, and fault tolerance are all important.

This paper describes a prototype web server written in Concurrent Haskell (with extensions), and presents two useful results: firstly, a conforming server could be written with minimal effort, leading to an implementation in less than 1500 lines of code, and secondly the naive implementation produced reasonable performance. Furthermore, making minor modifications to a few time-critical components improved performance to a level acceptable for anything but the most heavily loaded web servers.

A Typed Representation for HTML and XML Documents in Haskell *Peter Thiemann*

We define a family of embedded domain specific languages for generating HTML and XML documents. Each language is implemented as a combinator library in Haskell. The generated HTML/XML documents are guaranteed to be well-formed. In addition, each library can guarantee that the generated documents are valid XML documents to a certain extent (for HTML only a weaker guarantee is possible). On top of the libraries, Haskell serves as a meta language to define parameterized documents, to map structured documents to HTML/XML, to define conditional content, or to define entire web sites.

The combinator libraries support element-transforming style, a programming style that allows programs to have a visual appearance similar to HTML/XML documents, without modifying the syntax of Haskell.

Secrets of the Glasgow Haskell Compiler Inliner *Simon Peyton Jones and Simon Marlow*

Higher-order languages, such as Haskell, encourage the programmer to build abstractions by composing functions. A good compiler must inline many of these calls to recover an efficiently executable program.

In principle, inlining is dead simple: just replace the call of a function by an instance of its body. But any compiler-writer will tell you that inlining is a black art, full of delicate compromises that work together to give good performance without unnecessary code bloat.

The purpose of this paper is, therefore, to articulate the key lessons we learned from a full-scale "production" inliner, the one used in the Glasgow Haskell compiler. We focus mainly on the algorithmic aspects, but we also provide some indicative measurements to substantiate the importance of various aspects of the inliner.

Faking It: Simulating Dependent Types in Haskell *Conor McBride*

Dependent types reflect the fact that validity of data is often a *relative* notion by allowing prior data to affect the types of subsequent data. Not only does this make for a *precise* type system, but also a highly *generic* one: both the type and the program for each instance of a family of operations can be computed from the data which *codes* for that instance.

Recent experimental extensions to the Haskell *type class* mechanism give us strong tools to relativize types to other *types*. We may simulate some aspects of dependent typing by making counterfeit type-level copies of data, with type constructors simulating data constructors and type classes simulating datatypes. This paper gives examples of the technique and discusses its potential.

Parallel and Distributed Haskells *P.W. Trinder, H-W. Loidl and R.F. Pointon*

Parallel and distributed languages specify computations on multiple processors and have a computation language to describe the algorithm, i.e. *what* to compute, and a coordination language to describe *how* to organise the computations across the processors. Haskell has been used as the computation language for a wide variety of parallel and distributed languages, and this paper is a comprehensive survey of implemented languages. We outline parallel and distributed language concepts and classify Haskell extensions using them. Similar example programs are used to illustrate and contrast the coordination languages, and the comparison is facilitated by the common computation language. A lazy language is not an obvious choice for parallel or distributed computation, and we address the question of why Haskell is a common functional computation language.

Chapter 2

Implementations

2.1 GHC

Report by: *Simon Peyton-Jones*

The Team

Simon Peyton Jones, Simon Marlow, Julian Seward, Reuben Thomas, (with particular help recently from Marcin Kowalczyk, Sigbjorn Finne, Ken Shan)

2.1.1 Current status

In early October we released GHC 5.02. This is the first version of GHC that works really solidly on Windows, and it also has a much more robust implementation of GHCi, the interactive version of GHC. Compared to earlier releases our test infrastructure is in much better shape, and we were pretty confident about its reliability. Perhaps in response to this rash claim, lots of people started to use it and, sure enough, a significant collection of bugs were reported. So we will release GHC 5.02.1 early in November. *[this has just happened (ed)]*
Simon PJ has spent quite a bit of time on a new demand analyser that now replaces the old strictness analyser and CPR analyser. The new thing is much faster, and much smaller (lines of code) than the analysers it replaces. Hopefully a paper will follow.
Simon M wrote a new compacting garbage collector that reduces the amount of real memory you need to run big programs.
Ken Shan has heroically done a fine Alpha port of GHC.

2.1.2 Future plans

Sadly, Reuben leaves at the end of October, and Julian at the end of Feb 02, when the grant that funds them runs out. That will leave the two Simons on GHC duty. So the tempo of GHC activity will reduce; we have no new sources of money in our sights. GHC is, and remains, an open-source project, and we welcome contributions from others. (Thanks to Ken, Sigbjorn, Marcin, and others who have pitched in recently.)
So our current short-term objective is to get GHC into a really solid, robust state — rather than adding lots of new features. In particular, we plan to spend the autumn on

- gap-filling
- improve performance

2.1.3 Gap-filling

There is a never-ending task of filling in things that nearly work and but don't quite do it right. E.g. warning about unused bindings isn't quite right; generics are incomplete; derived read on unboxed types doesn't work; derived Read generates obscene amounts of code; and so on. This is a bit of a thankless task, and we're much more motivated to get on with things that are actually holding people up. So please tell us.

2.1.4 Performance

We have not paid serious attention to the quality of the code GHC produces, or the speed at which it produces it, or the space it eats (esp GHCi), for quite a while. So we're going to work on

- improving the profiling tools that come with GHC
- applying them to GHC itself (speed and space)

In particular, Sunwoo Park, a summer intern from CMU, built a prototype implementation of lag/drag/void profiling, and retainer profiling. We plan to integrate these into our main release.

2.1.5 New features

Having said we're not concentrating on new stuff, here are the things that are floating around in our brains. Vote now!

- Libraries – the new hierarchical libraries story is mostly implemented already. We'll complete this and make it part of the next major release.
- GHC.NET – a back end for GHC that targets Microsoft's .NET platform. This is pretty well advanced.
- Higher ranked types a la Odersky/Laufer.
- Views – Ralf has been bullying Simon
- Syntax for arrows – Ross has been bullying Simon
- Meta Haskell – Tim has been bullying Simon

Further reading:

`http://www.haskell.org/ghc/`

2.2 Hugs

Report by: *Johan Nordlander*
Project status: *maintenance mode, volunteers needed*
Hugs 98 is a small and fast interactive programming system, that offers an almost complete implementation of Haskell 98. Its main strengths are

- Short compilation times.
- Runs on many different platforms.
- Easily portable sources, written in C.
- An array of experimental extensions.

Hugs 98 is open source, and thus dependent on volunteering efforts for its development. In particular, Hugs isn't maintained or supported by OGI anymore. An active mailing list, and the Hugs cvs archive, can both be reached from haskell.org. The new FFI is partially supported in the current release. A new release (tentatively scheduled for Nov. 30), that will include hierarchical module names and the rearranged hslibs, is currently being put together by Sigbjorn Finne, Alastair Reid, Jeff Lewis, and Johan Nordlander.

Further reading:

`http://www.haskell.org/hugs/`
`http://haskell.org/mailman/listinfo/hugs-users`

2.3 nhc98

Report by: *Malcolm Wallace*
The particular strengths of the nhc98 compiler are portability, space-efficiency, close adherence to Haskell'98, and extensive tool support to help you to engineer better programs.

Tool support:

- The Hat tracing/debugging project is currently hosted in nhc98 – further details of the state-of-the-art are listed under the 'Tracing' topic.
- nhc98 still has the best heap-profiling support of any Haskell system, and its time-profiler was recently improved to make it more accurate in attributions of costs.
- nhc98 has an interactive development environment, much like Hugs and ghci, based on the hmake compilation manager. This was added about one year ago.

Future developments:

- nhc98 currently implements the same FFI as Ghc and Hugs. Recently, the standard FFI specification has undergone some minor syntactic changes, and the helper libraries have been finalised. We hope to incorporate these changes into nhc98 in the near future.
- The extended module namespaces proposal has been implemented in nhc98 for nearly a year. Now that the 'core' set of extended libraries is pretty-much agreed, we intend to package these with the compiler pretty soon.
- If anyone would like to contribute to the development of nhc98, there are some self-contained projects that need attention: porting to any 64-bit architecture; permitting non-simple contexts, higher kinds, multi-parameter type-classes, and other type-system hacking; etc.

Further reading:

`http://www.cs.york.ac.uk/fp/nhc98/`

2.4 Eager Haskell

Report by: *Jan-Willem Maessen*
Project status: *new project*
Compiles arbitrary Haskell programs, but happens to run them eagerly using resource-bounded execution. There should be no difference in observed program behavior—every Haskell program is a valid Eager Haskell program (except that our compiler doesn't yet cover all of Haskell 98—we're missing qualified names and field names).
Except for the missing bits of language and libraries, this is a real honest-to-goodness Haskell implementation. It's even easy to hack.

Goals:

Make it easy to write efficient programs (eg tail-recursive loops) in Haskell. Explore the efficiency tradeoffs between eager and lazy execution. Extract parallelism from ordinary haskell programs without annotating them.

Release double-oh-negative-integer:

Within weeks. Perhaps days.

People:

me. Thus the stealthy pace.

Further reading:

`http://csg.lcs.mit.edu/~earwig/eager-haskell.html`

Chapter 3

Language Extensions

3.1 Foreign Function Interface

Report by: *Manuel Chakravarty*

Version 1.0 of the Haskell 98 FFI Addendum is nearing publication. This version of the addendum only covers interaction between Haskell and C code in detail, but is designed to be easily extensible with support for other languages, such as C++ and Java. The current draft is available from
`http://www.cse.unsw.edu.au/~chak/haskell/ffi.ps.gz`
The document is complete and if approved in its current form on the FFI mailing list will be circulated on the main Haskell list for comments from the community. The functionality of the FFI as defined in the addendum is currently available in GHC and NHC98; however, there are still syntactic differences between the definition and those implementations. They are expected to be resolved in the near future.

Further reading:

`http://haskell.org/mailman/listinfo/ffi`

3.2 Hierarchical Module Namespace

Report by: *Simon Marlow*

Back in February of this year, Malcolm Wallace proposed an extension to Haskell to support a hierarchical module namespace, and gave some suggestions as to how Haskell might use the extended namespace. The original message is here:
`http://www.mail-archive.com/haskell@haskell.org/msg08187.html`
At the same time, a mailing list for discussion of the changes was set up, libraries@haskell.org. The archives are here:
`http://www.haskell.org/pipermail/libraries/`
This report details the current status of the proposal, and outlines what we've been up to on the mailing list. There is also an evolving document describing the current proposal; an HTML version can be found here:
`http://www.haskell.org/~simonmar/libraries/libraries.html`

The Language Extension

A minimal extension to Haskell 98 has been agreed, namely to allow '.' as a valid character in a module name. The extension is supported by released versions of GHC and nhc98, and will be supported by a future release of Hugs. The mapping from module names to file names, although not part of the Haskell 98 standard, will also be similar between the implementations. Further extensions have been suggested, such as those to allow importing or renaming of multiple modules simultaneously, but none has been settled on. We're waiting until we have more experience with using the hierarchical scheme before deciding what further extensions, if any, are necessary.

The Hierarchy

We settled on a rough outline of what the hierarchy should look like to a Haskell programmer. The guiding principle we agreed on was that the hierarchy should reflect functionality, so libraries with similar functionality should be grouped together. One consequence of this decision is that whether a library is "standard" or not should not be reflected in its name or its position in the hierarchy.

The Libraries

We came to the conclusion that there ought to be a single set of "core" libraries used by all the implementations, with sources kept in a single place. Several issues surrounding the core libraries, such as licensing, versioning, portability & stability have been discussed; the current working proposals are detailed in the working document (URL above).
Work has begun on constructing the core libraries. The current sources can be perused in the CVS repository, here:
`http://cvs.haskell.org/cgi-bin/cvsweb.cgi/fptools/libraries/`
The current status is that most of GHC's old hslibs libraries have been migrated into the new framework, with the exception of posix, edison, HaXml, Parsec and a few others. GHC runs with the new libraries, but the development version of GHC hasn't fully switched over to the new scheme yet - this is expected to happen before the next major release of GHC.

TODO Much remains to be done, and there's ample scope for contribution: providing implementations of libraries, designing interfaces to libraries, pointing out problems and inconsistencies in existing libraries, testing implementations, porting the core libraries to other compilers/platforms, and so on.

3.3 Non-sequential Programming

3.3.1 Concurrent Haskell

Report by: *Simon Marlow*

Concurrent Haskell is a set of extensions to Haskell to support concurrent programming. The concurrency API (Concurrent) has been stable for some time, and is supported in two forms: with a preemptive implementation in GHC, and a non-preemptive implementation in Hugs. The Concurrent API is described here:

`http://www.haskell.org/ghc/docs/latest/set/sec-concurrent.html`

There has been some recent activity concerning the interaction between concurrency and exceptions, the result being the asynchronous exception API provided by GHC:

`http://www.haskell.org/ghc/docs/latest/set/sec-exception.html#SEC-ASYNCHRONOUS-EXCEPTIONS`

A future goal is to specify and standardise the Concurrent Haskell extension as a Haskell 98 addendum.

3.3.2 GpH – Glasgow Parallel Haskell

Report by: *Phil Trinder*

GpH is a minimal, conservative extension of Haskell'98 to support parallelism. Experience has shown that it is particularly good for constructing symbolic applications, especially those with irregular parallelism, e.g. where the size and number of tasks is dynamically determined. The project has been ongoing since 1994, initially at Glasgow, and now at Heriot-Watt and St Andrews Universities.

GpH extends Haskell'98 with parallel composition: `par`. Parallel and sequential composition are abstracted over as *evaluation strategies* to specify more elaborate parallel coordination, e.g. `parList s` applies strategy `s` to every element of a list in parallel. Evaluation strategies are lazy higher-order polymorphic functions that enable the programmer to separate the algorithmic and coordination parts of a program. A number of realistic programs have been parallelised using GpH, including a Haskell compiler and a natural language processor.

GpH is publicly available from the page below, and is implemented on GUM, a sophisticated runtime system that extends the standard GHC runtime system to manage dynamically much of the parallel execution, e.g. task and data placement. GUM is portable: using C and standard communication libraries (PVM or MPI), and hence GpH is available on a range of platforms, including shared-memory, distributed memory and workstation clusters, e.g. Beowulf. GpH shares implementation technology with the Eden and GdH languages.

Current work includes making GpH architecture independent, i.e. deliver good parallel performance on a range of platforms. Improved parallel profiling, parallel semantics and abstract machines, and performance comparison with other languages.

Further reading:

`http://www.cee.hw.ac.uk/~dsg/gph/`

3.3.3 Glasgow Distributed Haskell

Report by: *Robert Pointon*

Robert Pointon, of Heriot-Watt University, has been working on Glasgow Distributed Haskell (GdH): GdH combines the multiple processes of Concurrent Haskell with the multiple processing elements of Glasgow Parallel Haskell (GpH). In summary the language is a minimal super-set of GpH and Concurrent Haskell and so maintains full backwards compatibility.

To support distribution we have only introduced the notion of "location":

- Location awareness - with explicitly placed I/O threads making use of the individual resources of each machine.
- Location independence - once created any location dependent object such as a thread or MVar can be used without worrying about where it is.

We have used GdH to write applications which include: a distributed file server, multiplayer games, and parallel skeletons. In terms of ongoing research, GdH is actively being used by the group here for looking at:

- The issues of lazy communication.
- Fault tolerance - recovering from failed remote computations.
- Mobile computing.

Oh, and the implementation is almost ready for public release!

Further reading:

`http://www.cee.hw.ac.uk/~dsg/gdh/`

3.3.4 Data Field Haskell

Report by: *Björn Lisper*
Project status: *dormant*

The continuing advances in semiconductor and hardware technology are leading to a situation where transistors are free and communication costly. This will make parallel systems-on-a-chip standard. These systems must be specified and programmed: this requires parallel programming and specification languages. The prevailing, process-parallel programming paradigms are however hard to master for many applications. Thus, efficient system and software development for these applications, on this kind of systems, will require simpler models on a higher level.

One such model is the data parallel model, which provides operations directly on aggregate data structures. These operations are often highly parallel. The data parallel model is particularly apt for data-intensive, computation-oriented applications like image and signal processing, neural network computations, etc. The Data Field Model is an attempt to create a formal, data parallel model that is suitable as a basis for high-level data parallel programming and specification. Data fields generalize arrays: they are pairs (f,b) where f is a function and b is a "bound", an entity which can be

interpreted as a predicate (or set). The model postulates some operations of bounds, with certain properties. Common collection-oriented primitives can be defined in terms of these operations, without referring to the actual form of the bounds. Data fields thus make a very generic form of data parallelism possible, where algorithms become less dependent on the actual data parallel data structure.
Data Field Haskell (DFH) is a dialect of the functional programming language Haskell that provides an instance of data fields. This language can be used for rapid prototyping of parallel algorithms, and for parallel high-level specification of systems. DFH provides data fields with "traditional" array bounds and with sparse bounds, infinite data fields with predicates as bounds, and data fields with cartesian product bounds. There is a rich set of operations on data fields and bounds. A forall construct, similar to lambda-abstraction, can be used to define data fields in a succinct and generic manner.
The current version of DFH extends Haskell 98. Its implementation is a modified version of nhc98 pre-release 19 (2000-06-05), originally from the functional programming group at York. Although much of DFH is defined in Haskell itself, a few crucial things aren't, so the implementation is not easily portable to other Haskell systems.
Currently, the project is dormant. One M. Sc. thesis project was recently carried out within the project: Data Field Haskell 98, where an earlier implementation of DFH was ported to Haskell 98.

Major Goals:

If this project is revived (or somebeody else picks up the thread), then the following is on the wish list:

- Case studies
- More portable architecture of the implementation (not relying on nhc)
- Some generalizations of the data fields provided by DFH
- Implementation of data field specific optimizations
- Parallel implementation

Further reading:

`http://www.mrtc.mdh.se/projects/DFH/`
`http://www.mds.mdh.se/~dal96asn/dfh98.htm`

This activity is a continuation of the Data Fields project at KTH, where also the first prototype implementation of Data Field Haskell was developed:
`http://www.it.kth.se/labs/paradis/project-datafields.html`

3.3.5 O'Haskell

Report by: *Johan Nordlander*
O'Haskell extends Haskell with support for monadic reactive objects and polymorphic subtyping. An implementation, O'Hugs, is available, which is an interactive programming system derived from Hugs 1.3b. O'Hugs also comes with reactive network programming APIs, and a fairly complete interface to the Tk graphical toolkit.
O'Hugs is maintained (although at a slow pace) by Johan Nordlander, Magnus Carlsson, and Björn von Sydow. New O'Haskell-related developments are currently directed towards the language Timber, which is a strict language with real-time capabilities that has inherited many of O'Haskell's features.

Further reading:

`http://www.cs.chalmers.se/~nordland/ohaskell/`
`http://www.cse.ogi.edu/PacSoft/projects/Timber/Default.htm`

3.3.6 pH (parallel Haskell)

Report by: *Jan-Willem Maessen*
An implicitly parallel dialect of Haskell, with provisions for side effects and special constructs for looping and for detecting termination. Uses a superset of the Eager Haskell parser, so the caveats about missing Haskell 98 features apply here too.
The ideas behind pH are best embodied in Arvind and Nikhil's book, "Implicit Parallel Programming in pH" (Morgan-Kaufman, 2001). There's a compiler release available, but it doesn't match the book as Nikhil never had the chance to make the necessary changes.
People: Alejandro Caro, myself, Arvind, Nikhil, Jacob Schwartz, Mieszko Lis, Lennart Augustsson, etc. I'm the only one of the above currently in academia, and I'm working full-time on Eager Haskell.

Further reading:

(I don't have write bits on this anymore, or I would have thrown most of it out and re-written it as it's so old)
`http://csg.lcs.mit.edu/projects/ph/`

3.3.7 Eden

Report by: *Rita Loogen and Steffen Priebe*
Eden extends Haskell by a small set of syntactic constructs for explicit process specification and creation. While providing enough control to implement parallel algorithms efficiently it frees the programmer from the tedious task of managing low-level details by introducing automatic communication (via head-strict lazy lists), synchronisation, and process handling.
Eden's main constructs are process abstractions and process instantiations. The expression `process x -> e` of a predefined polymorphic type `Process a b` defines a *process abstraction* mapping an argument `x::a` to a result expression `e::b`. Process abstractions of type `Process a b` can be compared to functions of type `a -> b`, the main difference being that the former, when instantiated, are executed in parallel.

Process instantiation is achieved by using the predefined infix operator `(#) :: Process a b -> a -> b`.
Higher-level coordination is achieved by defining higher-order functions over these basic constructs. Such *skeletons*, ranging from a simple parallel map to sophisticated replicated-worker schemes, have been used to parallelise a set of non-trivial benchmark programs.
Eden has been implemented by modifying the parallel runtime system GUM of GpH. Differences include stepping back from a global heap to a set of local heaps to reduce system message traffic and avoid global garbage collection. The current (freely available) implementation is based on GHC 3.xx. An Eden implementation based on GHC 5.xx will be available in the near future.
Eden has been jointly developed by two groups at Philipps Universität Marburg, Germany and Universidad Complutense de Madrid, Spain. The project has been ongoing since 1996.
Current and future topics include program analysis, skeletal programming, and polytypic extensions.

Further reading:

`http://www.mathematik.uni-marburg.de/inf/eden`

3.4 Type system extensions and variants

3.4.1 A General Type Class Framework based on Constraint Handling Rules

Report by: *Martin Sulzmann*
We use Constraint Handling Rules (CHRs) to describe various type class extensions. Under sufficient conditions on the set of CHRs, we have decidable operational checks which enable type inference and ambiguity checking for type class systems. Current work includes, the combination of open/closed-world style overloading and a general coherence result.
TIE, a CHR-based type inference engine, and the underlying CHR-solver have been implemented in Haskell (it's only a prototype yet, but we're working on extending TIE).

Further reading:

`http://www.cs.mu.oz.au/~sulzmann/chr/`

3.5 Generic Haskell

Report by: *Johan Jeuring*
Software development often consists of designing a datatype, to which functionality is added. Some functionality is datatype specific, other functionality is defined on almost all datatypes, and only depends on the type structure of the datatype. Examples of generic (or polytypic) functionality defined on almost all datatypes are the functions that can be derived in Haskell using the deriving construct, storing a value in a database, editing a value, comparing two values for equality, pretty-printing a value, etc. A function that works on many datatypes is called a generic function.
There are at least two approaches to generic programming: use a preprocessor to generate instances of generic functions on some given datatypes, or extend a programming language with the possibility to define generic functions.

3.5.1 Preprocessors

DrIFT (`http://www.cs.york.ac.uk/fp/DrIFT/`) is a preprocessor which generates instances of generic functions. It is used in Strafunski (`http://www.cs.vu.nl/Strafunski/`) to generate a framework for generic programming on terms.

3.5.2 Languages

PolyP (`http://www.cs.chalmers.se/~patrikj/poly/`) is an extension of a subset of Haskell in which generic functions can be defined and type checked. Polyp allows the definition of polytypic functions on a limited set of datatypes. Hinze has shown how to overcome some of the limitations of Polyp by extending Haskell with a construct for defining type-indexed functions with kind-indexed types. Generic Haskell (`http://www.generic-haskell.org/`) is based on Hinze's ideas. Also GHC has an extension that uses Hinze's idea to add derivable type classes to Haskell.

Current hot topics:

Generic Haskell: Type-indexed data types; different ways to define generic functions; defining generic functions on particular types, and particular constructors, XML tools as generic programs.
[Generic Haskell version 0.99 has just been released (ed)]

Major Goals:

Extend Generic Haskell with constructs for defining generic functions on particular types and constructors.

Further reading:

`http://www.cs.york.ac.uk/fp/DrIFT/`
`http://www.cs.chalmers.se/~patrikj/poly/`
`http://www.generic-haskell.org/`
`http://www.cs.vu.nl/Strafunski/`
There is a mailing list for Generic Haskell: generic-haskell@cs.uu.nl. See the homepage for how to join.

3.6 Other Developments

In other news (as they say on tv;-), Mark Shields and Simon Peyton Jones have taken another go at the topic of "First-Class Modules for Haskell":
`http://research.microsoft.com/Users/simonpj/papers/first-class-modules/index.htm`

From their abstract: "In this paper we refine Haskell's core language to support first-class modules with many of the features of ML-style modules. Our proposal cleanly encodes signatures, structures and functors with the appropriate type abstraction and type sharing, and supports recursive modules. All of these features work across compilation units, and interact harmoniously with Haskell's class system. Coupled with support for staged computation, we believe our proposal would be an elegant approach to run-time dynamic linking of structured code.
Our work builds directly upon Jones' work on parameterised signatures, Odersky and Laufer's system of higher-ranked type annotations, Russo's semantics of ML modules using ordinary existential and universal quantification, and Odersky and Zenger's work on nested types. We motivate the system by examples, and include a more formal presentation in the appendix. "

Chapter 4

Libraries

4.1 Foreign Function Interface

Report by: *Manuel Chakravarty*
Support for Haskell's Foreign Function Interface is separated into language extensions, support libraries built on top of these, and tools that make use of the libraries and extensions. The support libraries are covered in the draft Haskell 98 FFI Addendum, discussed in section 3.1 of this report.

4.2 Hierarchical Module Namespace

Report by: *Simon Marlow*
The language extension that permits hiearchical module namespaces was motivated by the need to organise the growing body of Haskell libraries, both user-contributed and those supported across Haskell implementations. See the subsections on "The Hierarchy" and "The Libraries" in section 3.2.

4.3 Graphical User Interfaces

4.3.1 GUI Library API Task Force

Report by: *Manuel Chakravarty*
Project status: *new project*

Goals

- Development of a GUI library API for Haskell that is portable across Haskell systems and operating/windowing systems
- The library focuses on graphical *user interfaces* (ie, buttons, menus, scrollbars, selection lists, etc) as opposed to drawing and animation routines.
- Verification of the design by at least one implementation of a library that exports the GUI API
- Minimal use of features that are not included in Haskell 98 (in fact, we currently believe that we can keep the API completely free of any non-H98 features; however, application programs that use the API will need some non-H98 functionality)
- While we do not require support of concurrency by Haskell systems implementing the API, the use and behaviour of the API must be well-defined on systems that do support concurrency
- Core library:
  - Due to the complexity of modern GUIs, we cannot hope to cover a fully-fledged GUI library in reasonable time and with reasonably effort; thus, we concentrate on a core library (what exactly constitutes the core remains to be defined)
  - The core library needs to provide a migration path to at least one fully-fledged GUI library
- Openness for tools support; for example, it should be possible to support graphical interface builders (such tools are not part of the API, but we need to ensure that we do not seriously complicate the construction of such tools)

Non-goals

- We will not design a fully-fledged GUI library from scratch: Just as a reference point, GTK+ has 93 different widgets (and that's just the base set) and probably over 1000 functions. Moreover, there are surely 100-200 different types of callbacks. I am pretty sure that Qt, the Win32 GUI API, and the MacOS are of similar size.

  This is well beyond the scope of what we can achieve with our resources.
- We will not design a "purely functional GUI":
  - The pragmatic reason: H98 textual I/O works via the IO monad; so what's the problem with having graphical I/O in the IO monad?
  - The state-of-the-art reason: There doesn't seem to be a clear favourite in the proposals for purely functional GUIs. Moreover, it has yet to be shown that any of the proposals scales to real-life GUI applications.

  We do not intend to solve sophisticated research problems here. We want to get a workable solution quickly. Always remember: Worse is Better `http://www.jwz.org/doc/worse-is-better.html`.

The Plan

- Handling of state altered by both the application and by GUI widgets:
  - The GUI API will be kept free of any commitment as to which mechanism to use here.
  - The simplest solution, for a single-threaded application, is to use 'IORef's and, for a multi-threaded application, is to use 'MVar's (or primitives that are defined in terms of these).
  - Advantages of this design:
    * Doesn't require Haskell systems that want to provide the GUI library to also support concurrency
    * Doesn't preclude the use concurrency (where it is available)
    * 'IORef's are very easy to implement if any system doesn't have them yet
    * More sophisticated approaches (that often require language extensions or are still experimental) can be implemented on top of this basic API - eg, FranTk, Yahu, Fruit, iHaskell, etc.
  - It is possible to write reasonably nice Haskell programs in this approach; for an example, see

    `http://www.cse.unsw.edu.au/~chak/haskell/gtk/BoolEd.html`

    (Note how many of the functions need to be in the IO monad anyway, because they need to perform file I/O.)
- Start from the API of GTK+ as a base line:
  - This is an API that has a proven track record of being suitable for small, medium, and large-scale applications
  - The API does not overly rely on object-oriented language features
  - A large body of free documentation exists for the API, which can be adapted to our purpose
  - A working Haskell binding exists that can be extended to provide a first implementation of the Haskell GUI API
  - A clear migration path exists for applications that need more sophisticated features than the core API provided by the Haskell GUI
  - GTK+ itself is already reasonably platform independent
- Ensure platform independence:

  The Haskell GUI will be restricted to GTK+ features that can be implemented on other major platforms with reasonable effort
- Haskell-ise the API:
  - C-ish functionality will be cut out
  - Full use will be made of Haskell's type system and, in particular, type classes
- Convenience libraries

  The API will include a set of convenience libraries on top of the basic API (eg, by providing layout combinators)

Comments

The effort by the GUI Task Force is *not* meant to preempt any other GUI efforts for Haskell. The goal of the GUI Task Force is very limited, so that we arrive at a workable solution quickly. It's purpose is merely to provide some baseline functionality that any Haskell user can rely on having available. The relation to other GUI projects can be as twofold:

- Simple bindings to GUI libraries (such as Gtk+HS) could provide the standard GUI API as one way to access the library.
- High-level approaches (such as FranTk or Yahu) could be implemented on top of the GUI API.

The combination of the two would lead to the nice situation where we can use different high-level APIs on different widgets sets.

Further reading:

`http://www.haskell.org/mailman/listinfo/gui`

4.3.2 Port of the Clean Object I/O library to Haskell

Back in February, Simon Peyton-Jones issued a rather unusual call to the Haskell mailing list, titled "A GUI toolkit looking for a friend". He was referring to a promising port of the well-known Clean Object I/O library to Haskell:
"Peter Achten, its author, spent a few weeks in Cambridge, porting the Clean library to Haskell. The results are very promising. The main ideas come over fine, translating unique-types to IO monad actions, and the type structure gets a bit simpler.
So what we need now is to complete the port. Peter didn't have time to bring over all the widgets, nor did he have time to clean up the Haskell/C interface. (Clean's FFI is not as well-developed as Haskell's, so the interface can be made much nicer.) The other significant piece of work would be to make it work on Unix, perhaps by re-mapping it to GTK or TkHaskell or something.
So the main burden of this message is:
Would anyone be interested in completing the port?
Fame (if not fortune) await you! The prototype that Peter developed in is the hslibs/ CVS repository, and the GHC team would be happy to work with you to support your work. (The more compiler-independent we can make the library, the better.) Peter Achten is willing to play consultant too. The Clean team are happy for the code to be open source – indeed, all the hslibs/ code is BSD-licensed.

It would not be the work of a moment. There are subtle issues involved (especially involving concurrency), and the design is not complete, so it isn't just boring hacking. So it should fun."
Krasimir Angelov has been the first volunteer to accept that challenge, with all the small print attached. He has been pretty active in the last few weeks:
"At this time the project is near its completion. The Haskell/C interface is completed. The library supports windows, dialogs and various kinds of controls. However, there are other items that are meant to be completed (menus, timers and other). It can already be used for simple GUI applications. My idea is not only to port the library, but also to extend it with various items.

- The library needs a backend that targets GTK. I also plan to extend the library with many new controls such as: ListBox, Grid, formatted edit controls and other different database oriented controls (I will use modified version of HaSQL).
- I want to bind ObjectIO and HOpenGL. This will allow the developing of graphic oriented applications.
- Similar to Delphi and Microsoft Visual Basic in ObjectIO every object has a list of its own attributes. That's why I think it is a good platform for developing a similar builder for Haskell. There is a need of good interactive development environment for Haskell and I think that one front-end of GHCi based on ObjectIO is a good alternative. This is my final aim.

Everyone is welcome to help, especially for porting to GTK. Current sources are in CVS repository.

Further reading:

CVS: `http://cvs.haskell.org/cgi-bin/cvsweb.cgi/fptools/hslibs/object-io/`

4.4 Graphics

4.4.1 HOpenGL – OpenGL Haskell Binding

Report by: *Sven Panne*
Project status: *Has been used by a handful users over the last two years and gains some momentum recently*
The goal of this project is to provide a binding for the OpenGL rendering library which utilizes the special features of Haskell, like strong typing, type classes, modules, etc., but still has the "flavour" of the ubiquitous C binding. This enables the easy use of the vast amount of existing literature and rendering techniques for OpenGL while retaining the advantages of Haskell over C. Portability in spite of the diversity of Haskell systems and OpenGL versions is another goal.
HOpenGL includes the simple GLUT UI, which is good to get you started and for some small to medium-sized projects, but HOpenGL doesn't rival the GUI task force efforts in any way. Smooth interoperation with GUIs like gtk+hs on the other hand *is* a goal.
The short-term objectives are ironing out the remaining small portability problems and enhance the packaging of the current distribution. HOpenGL has been reportedly tested on Intel-Linux, Windows 98, and Sparc-Solaris with OpenGL versions ranging from 1.0 to 1.2.1, but there are probably still some combinations which don't work smoothly yet.
A medium-term objective is more or less a rewrite of OpenGL. After some experimentation the best route is probably as follows: There is an official description of the OpenGL API (including all extensions)
`http://oss.sgi.com/projects/ogl-sample/registry/`
`http://oss.sgi.com/cgi-bin/cvsweb.cgi/projects/ogl-sample/main/doc/registry/specs/`
in the form of a specialized IDL from which a complete low-level binding could be generated automatically. A layer above this should make this a bit more Haskell-like. Currently a prototype which generates all data types including (un)marshaling functions already exists, but a translator for the API calls themselves has not been written yet.
Currently the coding is almost exclusively done by Sven Panne, but people are invited to join. Anyway, proposals for the user API (the 2nd layer mentioned above) and comments on the current API are much more urgent.
HOpenGL needs the new FFI and complex instance heads, but the latter non-H98 requirement is not really crucial and should be the topic of some debate.

Further reading:

`http://www.haskell.org/mailman/listinfo/hopengl`
`http://www.cin.ufpe.br/~haskell/hopengl/`

Chapter 5

Tools

5.1 Foreign Function Interface

5.1.1 C->Haskell

Report by: *Manuel Chakravarty*
Project status: *beta release*

C->Haskell is an interface generator that simplifies the development of Haskell bindings to C libraries. The current stable release is version 0.9.9, which is available in source and binary form for a range of platforms. There is a concise tutorial and the Gtk+HS binding shows that the tool is ready for serious use.

The most recent improvement is support for single-inheritance class hierarchies as they occur in C APIs that use a limited form of object-oriented design (this is currently available from CVS only, version 0.10.x). For the near future, simplified marshalling for common signatures as well as an example-based tutorial are planned. Updates on recent developments are available from the project homepage.

Further reading:

`http://www.cse.unsw.edu.au/~chak/haskell/c2hs/`

5.1.2 Java and Corba bridges

There have been some new or renewed activities in connecting Haskell to other languages recently. Two of these have their files at `http://sourceforge.net`.

The introduction to Zoltan Varga's Haskell-Corba interface says: "This software package implements an interface between the Haskell programming language and the MICO CORBA implementation. It allows Haskell programmers to write CORBA clients and servers in Haskell. It defines a language mapping from the IDL language used by CORBA to Haskell. It contains an IDL-TO-Haskell compiler which generates the necessary stub and skeleton routines from the IDL files.

The original version of this software was written in Clean as my Master's thesis. This package is inspired by Combat (formerly tclMico), which is a TCL-MICO interface program. Being the result of a thesis means that it is incomplete and emphasis was put on simplicity of implementation instead of performance.

The current version only works with MICO, but it is possible to port it to other ORBs. A port to ORBacus is partly done."

No separate information appears to be available for Ashley Yakeley's Haskell to Java VM Bridge, but the source code is in CVS at sourceforge.

Further reading:

`http://haskell-corba.sourceforge.net/`
`http://sourceforge.net/projects/jvm-bridge/`

5.2 Tracing and Debugging

Report by: *Olaf Chitil*

There have been and still are a large number of research projects on tracing lazy functional programs for the purpose of debugging and program comprehension. Most of these projects did not yield tools that can be used for Haskell programs in practice, but in the last few years the number of tracing tools for Haskell has increased.

Freja provides algorithmic debugging of Haskell programs but supports only a subset of Haskell and runs only on Sparc/Solaris. Hood is a portable library that permits to observe data structures at given program points. The February 2001 release of Hugs directly supports a variant of Hood, making observation of user defined data structures easier. GHood extends Hood by a graphical backend which can animate observations, giving insight into dynamic program properties (animations can be added to web pages). There are no concrete plans for further development of these systems in the near future.

The development of the algorithmic debugger Buddha (not currently available) is an ongoing research project.

The Haskell tracing system Hat is based on a multi-purpose trace file. The specially compiled Haskell program creates at runtime a trace file. Hat includes tools to view the trace in various ways: algorithmic debugging a la Freja; Hood-style observation of top-level functions; stack-trace on program abortion; backwards exploration of a computation, starting from (part of) a faulty output or an error message. Hat is developed within an active research project. Hat is currently integrated in nhc98 but in a few months a version that will work together with any Haskell compiler will be available. Hat shall enable tracing of any Haskell98 program, the few remaining language limitations will be lifted. It is already possible to invoke other viewing tools from some of the view-

ing tools but further integration and general improvement of the viewing tools is planned.

Further reading:

`http://www.haskell.org/libraries/#tracing`

5.3 Scanning and Parsing

5.3.1 Happy

Report by: *Simon Marlow*

Happy is very much in maintenance mode. It is heavily used in GHC and is relatively bug-free, but maintenance releases are still made occasionally. The latest release is 1.11 (September 2001). Happy's web page is at
`http://www.haskell.org/happy/`

Chapter 6

Applications, Groups, and Individuals

The format of this chapter is still in flux, and its potential usefulness became apparent too late to expect good coverage in the first edition of this report, but you might want to think about adding a bit about your own use of Haskell to the next edition, due in about six months.

6.1 Commercial Applications

6.1.1 Galois Connections, Inc.

Report by: *Andy Gill*

Haskell is Galois Connection's "not so secret" weapon. We use it to help meet the various demands of our clients in a number of ways:

1. **High Assurance Software Development**

   All clients want programs that do what they should be doing. Haskell gives us a significant head start towards achieving high assurance. Between leveraging the type system, writing programs that are concise enough to actually understand, and writing versions of client code that has the look-and-feel of a specification, we find Haskell a *practical* language to write our programs in.

2. **Domain Specific Languages**

   Domain Specific Language systems are just special purpose compilers, and Haskell excels at writing compilers.

3. **Translators**

   Galois recently had a project to help a client change how an API was used over a large code base. We wrote a translator, based on the SML/C-Kit, that did the translation using a type inference algorithm. OK, SML is not Haskell, but next time we'll use Haskell :-)

The basic idea behind Galois is solving difficult problems using functional languages. Furthermore, we believe that Haskell is the right language for handling the complex problems that arise in many parts of software engineering.

Further reading:

`http://www.galois.com`

6.2 Research Groups

Many research groups have already been covered by their larger projects in other parts of this report, especially if they work almost exclusively on Haskell-related projects, but there are more groups out there who count some Haskell-related work among their interests.

6.2.1 Functional Programming at Yale

Report by: *John Peterson*

The functional programming group at Yale is using Haskell and general functional language principals to design domain-specific languages. We are particularly interested in domains that incorporate *time flow*. Examples of the domains that we have addressed include robotics, user interfaces, computer vision, and music. The languages we have developed are usually based on Functional Reactive Programming (FRP). FRP was originally developed by Conal Elliott as part of the Fran animation system. It has three basic ideas: continous-time signals (behaviors), discrete-time signals (messages), and switching. FRP is particularly useful in hybrid systems: applications that have both continuous time and discrete time aspects.

FRP is a work in progress: there are many decision points in the FRP design space and we view FRP as a family of languages rather than a specific one. We have recently used arrows to build a new implementation of FRP that has a number of operational advantages. Although FRP has traditionally been implemented in Haskell, we have also been looking at direct compilation of FRP programs. We are particularly interested in compilation for resource-limited systems such as embedded controllers.

We have not yet released a version of FRP or our FRP-based languages such as Frob or FVision, but we expect to release software before the end of the year.

At present, the members of our group are Paul Hudak, John Peterson, Henrik Nilsson, Walid Taha, Antony Courtney, Zhanyong Wan, and Liwen Huang.

Further reading:

`http://haskell.org/frp`
`http://www.haskell.org/yale/`

6.2.2 Functional Programming Research Group at Kingston Business School (Kingston University)

Report by: *Chris Reade*
Application Area: Internet applications
Members:
(Kingston) Chris Reade, Dan Russell, Phil Molyneux, Barry Avery, David Martland
(British Airways) Dominic Steinitz
Contact: Dan Russell D.Russell@kingston.ac.uk
This is a relatively new community which has been developing internet applications using advanced language features (functional, typed and higher order). Part of our motivation is to investigate advantages of a functional approach to such application areas, but also to identify areas for further language and library development.
We have built an LDAP client with a web user interface entirely in Haskell (reported at the 3rd Scottish Functional Programming Workshop in August 2001). This is being further developed to include asynchronous processes (using Concurrent Haskell).
Over the next year we hope to provide libraries for the Haskell community to work in this area and to attract funding to expand the research.

Further reading:

FP Group: `http://www.kingston.ac.uk/~bs_s075/Research/fpres.html`
Chris Reade: `http://www.kingston.ac.uk/~bs_s075`

6.2.3 Functional Programming Research at UKC

Report by: *Claus Reinke*
Here at the University of Kent at Canterbury, about half a dozen people pursuing research interests in functional programming have formed a functional programming interest group. Our projects are not limited to Haskell, so not all of them are mentioned here, but there are still quite a few Haskell-related activities:
Keith Hanna is working on bringing together the intuitive graphical interface of spreadsheet-like systems with the expressiveness and type-security of Haskell. A prototype system, named Vital, and an overview paper are available. Stefan Kahrs is interested in the boundaries of type system expressiveness, and has been looking at what one can or cannot do with Haskell types & classes. Chris Ryder's current topic are software metrics for Haskell programs and their visualisation. Simon Thompson, apart from producing educational material to help others learn Haskell (such as "The Craft of Functional Programming"), is working mostly where logic, types, programming, and verification come together.
Tony Daniels still looks at the semantics of time in Fran every now and then. Leonid Timochuk has been working on a Haskell implementation of Aldor--, a functional subset of the dependently typed Aldor language, originally developed for the purpose of computer algebra. Claus Reinke (yours truly), after a stint in the visualisation of Haskell program observations (GHood), has been trying to bring together virtual worlds (in the form of the standard Virtual Reality Modeling Language VRML'97) and functional programming (Haskell, with some FRP ideas) in a project named FunWorlds. More recently, he has also been seen chasing Haskell Community reports. Axel Simon has just joined us on one of the positions we advertised in the Job Adverts part on haskell.org.
In latest developments, Simon and Claus have been investigating the potential for *refactoring functional programs*. Refactoring means changing the structure of existing programs without changing their functionality, and has become popular in the object-oriented and extreme programming communities as a means to achieve continuous evolution of program designs. We want to explore the wealth of functional program transformation research to bring refactoring to Haskell programmers. **We have just received confirmation of funding and will be advertising for a post-doctoral researcher soon, but if you are interested, please get in touch with us now!**

Further reading:

FP group (informal page;-): `http://www.cs.ukc.ac.uk/people/staff/acd/fp-group.html`
Haskell metrics: `http://www.cs.ukc.ac.uk/people/rpg/cr24/medina/`
GHood: `http://www.cs.ukc.ac.uk/people/staff/cr3/toolbox/haskell/GHood/`
FunWorlds: `http://www.cs.ukc.ac.uk/people/staff/cr3/toolbox/haskell/FunWorlds/`
Vital: `http://www.cs.ukc.ac.uk/people/staff/fkh/Vital/`
Some initial info about Refactoring Functional Programs: `http://www.cs.ukc.ac.uk/people/staff/sjt/Refactor/index.html`

6.3 Individual Haskellers

As it turns out, many Haskellers do not currently have the benefit of being in a large group of like-minded people. Following the large numbers of students being introduced to Haskell, this group of individuals around the world might well be the largest group of Haskell users and, in fact, many of those students who decide that knowing Haskell is a skill too useful to forget might find themself isolated after leaving their university. As Hal Daume suggests:
"It seems to me that many people are in this situation, which is rather unfortunate. If not only for the ability to walk down the hall and ask someone if they could look over my code. One thing that may perhaps be useful would be to identify serious Haskellers (say people with >10k LOC in Haskell under their belts) who happen to be the only people in their organizations who use Haskell and try to form little groups of maybe 5 people with similar research (or applications).

This would probably cut down on the "what's wrong with my code" posts to the mailing lists and would also give a more personal avenue for discussing issues (I know that personally, since I deal with tons of data in large files, memory management issues, strictness issues, etc. are of prime concern. Other people might have more problems related to, say, multiparameter type classes and whatnot, if their field is more in that direction – hard to say).

Anyway, that's just off the top of my head...I don't know whether it would actually be useful...one of the niceties about having someone down the hall is you can concurrently look at the code and find the problem (or the necessary optimization, as is often my desire). Whether something like this could work online, I don't know."

Well for a start, here are brief statements by the first few Haskellers to respond to my very late call for "micro-reports", in the hope of finding other Haskellers working in related areas. I hope this section will expand in future reports, and that the Haskell community finds other good ways to support its members. The main Haskell mailing lists (haskell, haskell-cafe) are certainly a good place to start organising more local (or networked, smaller) Haskell interest groups. In some cases, a re-organisation of the current mailing lists might also help - I could imagine a list on optimising and profiling (tools, techniques, and problems). Also, the currently rather inactive group on debuggers could become more lively if it widened its scope to debugging (again, covering tools, techniques, and problems).

The idea here, as in the earlier sections, is to let others know what you are working on, so that Haskellers with related interests can find together for the purposes of cooperation or technical discussions.

Hal Daume (`http://www.isi.edu/~hdaume/`) is currently a first year PhD student in Computer Science at the University of Southern California: "My research interests are in the area of computational linguistics, which is, naively, the study of getting computers to understand natural languages (like English). I'm currently using Haskell exclusively to do statistical natural language processing research applications (mostly in summarization and aggregation)."

John Heron (jheron@enteka.com) has been working sporadically on a couple of projects: "None of them are at a stage where there's a whole lot more than talk, but I've done some of thinking about them:

NetInfer takes a router topology expressed in an adjacency matrix and Cisco router configurations. Using this information it infers network reachability information for static routes and RIPv1&2.

dbkit is an interactive, in-memory implementation of Codds Relational Algebra and Tuple Calculus. Initially based on Andrew Rock's RelationalDB module, the emphasis will be on finding an formulation which reflects the theoretical definitions clearly.

Based on my schedule between now and the end of the year, I expect that I can have these two bits working by the end of the year. At that point, perhaps I can spark some interest in the community in helping me out. For now, just talking about them publicly, and public speech's implied burden of clarity is most of the help I need."

John also has some longer-term visions attached to these concrete projects. Check out his projects page at
`http://jheron.best.vwh.net/projects.html`