# Haskell Communities and Activities Report

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

*– third edition –*

*November 11, 2002*


Claus Reinke (editor), University of Kent at Canterbury, UK
Krasimir Angelov, Bulgaria
Sengan Baring-Gould, National Semiconductor Corporation
Andrew Bromage, RMIT University, Melbourne, Australia
Manuel Chakravarty, University of New South Wales, Australia
Olaf Chitil, The University of York, UK
Duncan Coutts, Oxford, UK
Joe English, Advanced Rotorcraft Technology, Inc., USA
Levent Erkok, OGI School of Science and Engineering, OHSU, USA
Andre W B Furtado, Federal University of Pernambuco, Brazil
Murray Gross, City University of New York, USA
Kevin Hammond, University of St Andrews, Scotland
Dean Herington, University of North Carolina at Chapel Hill, USA
John Hughes, Chalmers University, Sweden
Patrik Jansson, Chalmers University, Sweden
Johan Jeuring, Utrecht University, The Netherlands
Jose Emilio Labra Gayo, University of Oviedo, Spain
Ralf Lämmel, VU and CWI, Amsterdam, The Netherlands
Rita Loogen and Steffen Priebe, University of Marburg, Germany
Christoph Lüth, George Russell, and Christian Maeder, University of Bremen, Germany
John Meacham - California Institute of Technology, Alum., USA
Jan-Willem Maessen, Massachusetts Institute of Technology, USA
Simon Marlow, Microsoft Research Cambridge, UK
John Peterson, Yale University, USA
Rex Page, Oklahoma University, USA
Ross Paterson, City University London, UK
Simon Peyton Jones, Microsoft Research Cambridge, UK
Bernie Pope, University of Melbourne, Australia
Chris Reade, Kingston University, UK
Alastair Reid, Reid Consulting (UK) Ltd., UK
Chris Ryder, University of Kent at Canterbury, UK
Uwe Schmidt, Fachhochschule Wedel, Germany
Axel Simon, University of Kent at Canterbury, UK
Satnam Singh, Xilinx Inc, USA
Doaitse Swierstra, Utrecht University, The Netherlands
Martin Sulzmann, National University of Singapore, Singapore
Peter Thiemann, University of Freiburg, Germany
Phil Trinder, Heriot Watt University, Scotland
Eelco Visser, Utrecht University, The Netherlands
Malcolm Wallace, The University of York, UK
Ashley Yakeley, Seattle WA, USA

## Preface

Let me start with an early reminder, for the few of us who to whom these 6-monthly reports still come as such a surprise that they have problems scheduling the time they would need to contribute (perhaps even before the target deadline?-) – because there are a few summaries I had hoped for that have failed to materialize...

---

*Less than 6 months left until the next edition! In April 2003,* **you** *will be invited to contribute to the May 2003 edition of this report!*

---

In spite of such organisational difficulties, this third edition of the Haskell Communities and Activities Report is the biggest yet – even though many regular contributors have already agreed to condense the descriptive parts of their summaries to make room for new developments. So, while our first editions necessarily had the form of activity snapshots, having to introduce the groups and projects out there, we are now moving towards collecting updates relative to previous editions. Projects mentioned here for the first time will still have a bit more space for an introductory description.

For those who keep track of anniversaries: it is just a few weeks and 15 years after Haskell's conception in a meeting "held at the conference on Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon" (so the language report tells us), and this edition of our report shows the Haskell community being more active than ever, busily pursueing and realising the goals set out by the first Haskell committee.

A look at this report's table of contents give us a hint that much of the new activity is coming from applications of Haskell, in research and industry, by groups and by individuals (chapter 6). These applications need tools (chapter 5) and libraries (chapter 4), and the Haskell community is also working to develop those. And all this activity is in addition to continued development of Haskell implementations (4 reported in this edition; chapter 2), and it has not stopped experimental work on language extensions (chapter 3), either. On the contrary, development of implementations and language extensions is now often driven by the needs of applications, tools, and libraries, and oft-used extensions are well on their way of being supported across major implementations.

Even as the (now thoroughly;-) revised Haskell 98 report is being readied for publication (section 1.2), the community is looking for more pragmatic approaches to language standardisation. Instead of trying to do everything at once, language extensions are now permitted to ripe separately, as addenda to the report. Addenda specifying practically indispensible language extensions such as the foreign function interface (section 3.1) or the hierarchical namespaces (section 3.2) are converging on standardisation, and implementations are tracking the release candidates of the specifications or extension proposals. No longer is there a single Haskell committee working on everything, instead such extensions are now driven by separate interest groups of implementers and users. Discussions take place on open mailing lists, and release candidates and prototype implementations are presented to the main community for review.

Overall, it is an exciting time for Haskell development, and most of it driven by the efforts of volunteers, of which there can never be enough – a big thanks to all of you who are contributing to this drive in some way. Which brings us back to this report: just as users would like to keep up to date with implementations, tools, and libraries, implementors like to hear about all the interesting things that Haskellers feel enabled to do using these implementations, tools, and libraries. And non-Haskellers interested in the language would like to know what this is all about, and seeing that they would not be on their own if they decided to give Haskell a whirl in their next project should make that decision easier!

And so these 6-monthly overviews/ snapshots/ updates of Haskell communities and activities try to help by reflecting and summarizing some of what is going on out there, thanks to those of you who have contributed summaries to this report. Each summary introduces 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. By now, the HC&A reports are established additions to the more permanent `haskell.org` and the more fleeting various Haskell-related mailing lists.

But now you must be eager to read it, so – enjoy!-)

Claus Reinke,
University of Kent at Canterbury, UK

# Contents

# Chapter 1

# General

## 1.1 Haskell.org

**Report by:** *John Peterson*

Haskell.org belongs to the entire Haskell community - we all have a stake in keeping it as useful and up-to-date as possible. Anyone willing to help out at haskell.org should contact John Peterson (peterson-john@cs.yale.edu) to get access to this machine. There is plenty of space and processing power for just about anything that people would want to do there.
What can haskell.org do for you? There are a lot of things we can do that are of use to members of the haskell community:

- Advertise your work: whether you're developing a new application, a library, or have written some really good slides for your class you should make sure haskell.org has a pointer to your work.
- Hosting: if you don't have a stable site to store your work, just ask and you'll own haskell.org/yourproject.
- Mailing lists: we can set up a mailman-based list for you if you need to email your user community.
- Sell merchandise: give us some new art for the cafepress store. Publicize your system with a T-shirt.

The biggest problem with haskell.org is that it is difficult to keep the information on the site current. At the moment, we make small changes when asked but don't have time for any big projects. Perhaps the biggest problem is that most parts (except the wiki) cannot be updated interactively by the community. There's no easy way to add a new library or project or group or class to haskell.org without bothering the maintainers. The most successful sites are those in which the community can easily keep the content fresh. We would like to do something similar for haskell.org.
Just what can you do for haskell.org? Here are a few ideas:

- Haskell programmers are not graphic designers. Just about anyone could make haskell.org look nicer and more professional.
- Make the site more interactive. Allow people to add new libraries, links, papers, or whatever without bothering the maintainers. Allow people to attach comments to projects or libraries so others can benefit from your experience. Help tell everyone which one of the graphics packages or GUI's or whatever is really useful.
- Install a better wiki and move the current content over there.
- Develop a system where the pages for haskell.org live in a CVS repository so that we can more easily share out maintenance.
- Add searching capability to haskell.org.
- Take over the cafepress store and get more merchandise in there.
- Come up with better spam defenses for our mailing lists.

Some of these ideas would be good student projects. Be lazy - get students to do your work for you.

**Further reading:**

`http://www.haskell.org`
`http://www.haskell.org/mailinglist.html`

## 1.2 Revised Haskell 98 Report

**Report by:** *Simon Peyton Jones*

The good news is that the Haskell 98 Report is finally done. The original Haskell 98 report came out in February 1999. Soon afterwards, in 2000, I began to collect "typos". It soon became apparent that more than just typographical errors were involved. My goals became:

- Correct actual errors.
- Resolve inconsistencies.
- Resolve ambiguity and under-specification.
- Improve explanations.
- With extreme reluctance, make "improvements". (The biggest example of this is the new `genRange` method in the `Random` class.)

The Haskell committee *per se* no longer existed, so I have consulted the Haskell mailing list about every change.
Two years on, I have accumulated over 2000 email messages, almost all of which have required careful reading on my part. They led to more than 230 individually-documented changes, plus dozens of more minor corrections. At times it seemed that each time I put out a draft, people would spot a new raft of issues, but the process does now seem to have converged.

I am still making tiny changes but, from a content point of view, it's all over bar the shouting. A *lot* of people have now looked hard at the Report and, for all its flaws, it's now in pretty good shape.

### 1.2.1 Publication

It would be a very good thing for our community if the Report (both language and libraries) was available as a physical book. I made enquiries with several publishers, who declined politely. (They want to print high-volume undergraduate texts.) However, Cambridge University Press have agreed to publish it! It will come out both as Volume 13(1) (Jan 2003) of the Journal of Functional Programming, and as a separate book. It is being typeset now.

### 1.2.2 Copyright

The question of copyright remains open at the time of writing. CUP normally take copyright of what they publish, but the Haskell report is rather different, because it belongs to all of us, not just to me. What is definitely agreed is that the Report will continue to be available online, and that it can freely be reproduced for non-commercial purposes. But is that enough? The Haskell workshop was a good opportunity to have an open discussion about this question. There was a strong sentiment that we would much prefer the Report to be entirely unrestricted (as it has been to date). For example, would Debian distribute the Report as they do now? Probably not. Would the just-pre-publication online version be completely unrestricted? Presumably so. Would electronic distribution be OK? Unclear. And so on.
On the other hand, there was also strong feeling that publishing a book would be a Really Good Thing; that CUP will probably lose money on the exercise; and that they are being much more flexible already than their normal terms. (John Reppy tells me that the Standard ML Basis Library book is much more restricted.)
In the end, we took a straw poll, to get the sense of the meeting. Here's what happened.

| | | |
|---|---|---|
| (A) | 30 | Would like there to be a book, even if we have to accept that commercial reproduction would require negotiation with CUP. |
| (B) | 14 | Cannot accept (A) but would accept that commercial *non-electronic* reproduction would be at CUP's discretion. |
| (C) | 7 | Cannot accept (A) or (B): publish a book only if there is no limit whatsoever on reproduction. |
| | 18 | abstain |
| | 69 | TOTAL |

While many people have strongly-held views, it was a polite debate that generated at least as much light as heat. While there isn't a consensus of opinion about A/B/C, there was a consensus that views can legitimately differ on this point, and we should go with a majority view. So the outcome was that I should present as strong a case as possible to CUP for completely open publication; failing that, for open electronic publication; but failing that, publish anyway. This I have done, and CUP are considering it as I write.
I hope this is acceptable to the Haskell community. The Haskell workshop is not everyone with a stake, by any means, but there were enough people present to be reasonably representative.

**Further reading:**

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

## 1.3 Tips, Tricks, Tours and Tutorials

Some Haskellers have documented their own hard-won experience to help others. They have been working on web pages, short papers, tours, and tutorials touching on introductory examples of monads&co, giving guided tours and explanations of prelude, libraries & syntax, or tips about programming and resource tuning, even explaining the internals of GHC (scary;), or interpreting Hugs error messages.
Ultimately, all these things should be linked from the Haskell bookshelf (have another look, it is not limited to books:):
`http://www.haskell.org/bookshelf/`
Following on from our last edition, this section tries to enhance the visibility of such valuable resources, by introducing new additions.

---

Hal Daume mentions the "Yet Another Haskell Tutorial" project: `http://www.isi.edu/~hdaume/htut/`
"The goal of Yet Another Haskell Tutorial is to provide a complete intoduction to the Haskell programming language. It assumes knowledge neither of the Haskell language nor of functional programming in general. However, general familiarity with programming concepts (such as algorithms) will be helpful. This is not intended to be an introduction to programming in general; rather, to programming in Haskell."

---

Alastair Reid is working on a tutorial for the new Foreign Function Interface
`http://reid-consulting-uk.ltd.uk/docs/ffi.html`
"The goal is to collect together all the useful resources, compare the various ffi tools and implementations, provide tips and tricks for dealing with common awkward cases, etc. but the reality falls far behind the dream at present."

---

For those always-on Haskellers who can find spare moments between following the dozens of Haskell mailing lists, Andrew Bromage points out that there is a semi-official Haskell IRC channel on the freenode network. You can get there by pointing your IRC client at `irc.freenode.net` and joining `#haskell`. Logs are available here:
`http://tunes.org/~nef/logs/haskell/`

## 1.4 Haskell-related Publications

In this section, we try to give pointers to and perhaps short descriptions of recent Haskell-related publications (books, conference proceedings, special issues in journals, PhD theses,

etc.), with brief abstracts. For a more exhaustive overview of Haskell publications, see Jim Bender's "Online Bibliography of Haskell Research" (`http://haskell.readscheme.org`). Please make sure to keep him up to date about new (and not so new) Haskell-related publications!

And if you still haven't come across the Haskell bookshelf, you'll find it at `http://www.haskell.org/bookshelf/`. It lists textbooks, papers (especially of tutorial nature), proceedings of the "Advanced Functional Programming" summer and spring schools, as well as reference material, often created in the context of Haskell courses (see also our tips&tricks section 1.3).

---

In case you hadn't noticed, the JFP special issue on Haskell has finally appeared: Volume 12 – Issue 05 – July 2002 (we had it's abstracts in our November 2001 edition;-). If you or your institution has a subscription, you can also get it online via the journal's home page `http://www.dcs.gla.ac.uk/jfp/`, otherwise check out your nearest University library.

The 2002 Haskell Workshop proceedings are now online, in the ACM digital library, see the workshop home page for link and titles: `http://www.cse.unsw.edu.au/~chak/hw2002/`.

Functional and Declarative Programming in Education, FDPE02, was a workshop at PLI02, Pittsburgh. Many of its papers are relevant to Haskell based courses, and the proceedings are available at `http://www.informatik.uni-kiel.de/~mh/publications/reports/fdpe02/`.

---

"Value Recursion in Monadic Computations", *Levent Erkok*, PhD Thesis , OGI/OHSU, October 2002 (Advisor: John Launchbury).

This thesis addresses the interaction between recursive declarations and computational effects modeled by monads. More specifically, we present a framework for modeling cyclic definitions resulting from the values of monadic actions. We introduce the term "value recursion" to capture this kind of recursion.

Our model of value recursion relies on the existence of particular fixed-point operators for individual monads, whose behavior is axiomatized via a number of equational properties. These properties regulate the interaction between monadic effects and recursive computations, giving rise to a characterization of the required recursion operation. We present a collection of such operators for monads that are frequently used in functional programming, including those that model exceptions, non-determinism, input-output, and stateful computations.

In the context of the programming language Haskell, practical applications of value recursion give rise to the need for a new language construct, providing support for recursive monadic bindings. We discuss the design and implementation of an extension to Haskell's `do`-notation which allows variables to be bound recursively, eliminating the need for programming with explicit fixed-point operators.

Details (including downloadable text of the thesis) are available at: `http://www.cse.ogi.edu/PacSoft/projects/rmb` (see also section 3.7.1)

---

"A Formal Specification of the Haskell 98 Module System"
*Iavor S. Diatchki, Mark P. Jones, and Thomas Hallgren*

Many programming languages provide means to split large programs into smaller modules. The module system of a language specifies what constitutes a module and how modules interact. This paper presents a formal specification of the module system for the functional programming language Haskell. Although many aspects of Haskell have been subjected to formal analysis, the module system has, to date, been described only informally as part of the Haskell language report. As a result, some aspects of it are not well understood or are under-specified; this causes difficulties in reasoning about Haskell programs, and leads to practical problems such as inconsistencies between different implementations. One significant aspect of our work is that the specification is written in Haskell, which means that it can also be used as an executable test-bed, and as a starting point for Haskell implementers.

Available at `http://www.cse.ogi.edu/~diatchki/hsmod/`

# Chapter 2

# Implementations

## 2.1 The Glasgow Haskell Compiler

**Report by:** *Simon Peyton-Jones*

Simon Marlow and Simon Peyton Jones continue to hack away on GHC, aided by many others (see `http://www.haskell.org/ghc/contributors.html`). In the last six months we have been working on three major new areas:

1. The new hierarchical libraries are now fully available. As Haskell has become more widely used, the flat module namespace of Haskell 98 has proved increasingly troublesome. Using a hierarchical namespace for modules (a la Java) is a straightforward enough idea, but the idea has to be populated with an actual module hierarchy, and the agreed hierarchy has to be populated with real, properly documented implementations. The new module system has in turn had some impact on GHC's package mechanism (a way to group modules together for distribution purposes), and that is still setting down. At all events, GHC 5.04 came out with the new libraries as the primary library base. (Hugs and NHC are moving in the same direction, incidentally.)

2. Template Haskell (see section 3.6.1) led to a heart-and-lung transplant for GHC. Template Haskell lets you write meta-progams: Haskell code that runs at compile time and generates Haskell code, which is then compiled. Generating instance declarations from data type declarations is a typical application, but there are many others. Template Haskell forces a more intimate coupling of GHC's renamer (which establishes lexical scopes) and the typechecker than was previously the case. No difficulty in principle, but a lot of new plumbing was required.

   Template Haskell isn't in any released GHC, but it is available and working in the HEAD, for those who care to check out source code.

3. GHC has always used the *push-enter* evaluation model, in which function application goes like this: push the arguments on the stack, and then enter the function (which might be a thunk). The alternative is the *eval-apply* model: evaluate the function and call the resulting function closure, passing the (correct number of) arguments. Both eval-apply and push-enter can be used for both call-by-need and call-by-value languages, but in practice only lazy languages use push-enter.

   Push-enter looks very attractive for lazy languages, but it has many small costs scattered through the code generator and run-time system. For one thing, it seems to be practically impossible to compile it into `C--`, even though `C--` was designed to be as flexible a code generator as possible (`http://www.cminusminus.org`): we just couldn't think of a clean way to design `C--` to deal with push-enter. (Except by using `C--` in the unsatisfactory way we currently use C, namely ignoring the C stack and using an explicitly-managed stack instead.)

   So we have spent quite a bit of effort rejigging GHC's back end and run-time system to use eval-apply instead of push-enter. So far it seems that performance is pretty much unchanged; the number of lines of code in the runtime system and code generator is pretty much unchanged; but the complicated stuff is concentrated in a few places rather than being thinly distributed. And it makes the native `C--` route possible. This stuff isn't even in the repository yet, but it will be.

Other excitements

- On the type-system front, as well as big headline things (like functional dependencies or implicit parameters) GHC has accumulated quite a useful set of simple generalisations to the Haskell type system, not all of which are widely known. For example:
  - Infix type constructors
    `data a :+: b = Inl a | Inr b`
  - Type synonyms can contain foralls
    `type Foo b = forall a. (a,b) -> (b,a)`
  - A forall can appear to the right of an arrow
    `f :: b -> Foo b` In this case the forall is 'lifted' to the front; it is just as if you had written `f :: forall a b. b -> (a,b) -> (b,a)`
  - A forall can appear to the left of an arrow
    `h :: (forall a. a->a) -> Int` This can happen arbitrarily nested; arbitrary-rank polymorphism, not just rank 2.
  - Data type declarations can be explicitly kinded, which is useful when kind inference does not do the Right Thing `data T (a :: *->*) = T Int`

  On their own many of these features look a bit unnecesary, but Haskell encourages virtuoso programming with

types, and it can make all the difference to be able to say what you mean in the type language.

- GHC can now read, as well as write, External Core files. The idea is to make it easier for people to use GHC as a front end (generating External Core) or a back end (consuming External Core) or both. External Core is still a bit clunky, and we welcome suggestions for improving it.
- Recursive `do` notation, also known as '`mdo`', is implemented (see section 3.7.1). Implicit parameters are now bound with "`let`" or "`where`", not "`with`"; and that includes the "`let`" bindings in list comprehensions and `do`-notation.
- "Rebindable syntax" works for `do`-notation as well as numerics. The idea here is that when you say `-fno-implicit-prelude` all the numeric and `do`-notation syntax uses whatever `(+)`, `fromInteger`, `(>>=)`, `return`, etc are in scope, rather than using the `Prelude` functions (which the language report specifies). This allows you to completely replace the numeric or monadic infrastructure, which you just can't do in Haskell 98.
- Rewrite `RULE` pragmas have a more flexible way to specify in which compiler phases they apply.
- Interface files are stored in binary format now. To read them you have to use `ghc --show-iface`. At the same time, the format of "`.hi-boot`" files (used for breaking recursive-module loops) has been made more like regular Haskell and less like machine code (and hence more stable).

**Further reading:**

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

## 2.2 Hugs

**Report by:** *Alastair Reid*

**Team / status**

The Hugs98 interpreter is now maintained by Sigbjorn Finne and Jeffrey Lewis, both of Galois Connections, with help from Alastair Reid of Reid Consulting and Ross Paterson of City University London and others.
At the time of writing, a new major release is on the brink of being released. Feature highlights of this new release will be:

- Much improved FFI support (contributed by Alastair Reid), implementing almost all of the Haskell FFI specification.
- Support for the new Haskell hierarchical library specification (contributed by Johan Nordlander).
- Adoption of a significant subset of GHC's hierarchical libraries (contributed by Ross Paterson.)
- An (allegedly) complete implementation of the Haskell98 module system (Sigbjorn Finne).
- Numerous bug fixes since the previous major release in Dec 2001.

**Future plans**

Our primary goal is for Hugs to continue its move to greater compatability with Haskell98, GHC and NHC. This will be helped enormously by our mutual support for the FFI and hierarchical library specification and adoption of a common codebase for the libraries.
This release adds a lot of new functionality whilst maintaining compatibility with previous releases; future releases will drop some of this backwards compatibility.

**Further reading:**

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

## 2.3 nhc98

**Report by:** *Malcolm Wallace*

**Current Status**

We released nhc98 version 1.14 in June 2002, and hmake 3.06 in August. Since both the language and the compiler are now very stable, these were mainly bugfix releases. One new feature is a 'package' mechanism closely based on the ghc model, so third-party libraries can be easily added and removed. (hmake's handling of packages has also improved significantly.) Platform support now includes MacOS-X, in addition to just about every other Unix-like environment.

**Future Plans**

The next release (1.16) will probably arrive towards the very end of the year. Its main new features will be support for the latest most stable version of the standard FFI, and for the new library packages already supported by ghc and Hugs, both of which we have been promising for a long time now.

**Further reading:**

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

## 2.4 Eager Haskell

**Report by:** *Jan-Willem Maessen*

The Eager Haskell compiler runs ordinary Haskell programs using resource-bounded eager evaluation. Project details are available from
`http://www.csg.lcs.mit.edu/projects/index.php?link=eH.txt`
The best overview of the current system can be found in my recent Haskell Workshop paper:

http://www.csg.lcs.mit.edu/~earwig/
haskell-workshop.pdf

**Status:**

The Eager Haskell compiler is now available as source code runnable on Linux x86. It's already in use by about 15 MIT students taking Arvind's course. Installation should only require gcc 2.95.3 or later; hacking the compiler itself will require a working Haskell 98 compiler. See the project homepage for more information and a download link. Porting should be extremely easy if your system has gcc; x86 Linux is simply the only machine we've tested on.
The present compiler unifies the Eager Haskell and pH compilers under a single umbrella; the language in use can be selected by a compile-time switch. By default, phc is an ordinary haskell compiler.

# Chapter 3

# Language Extensions

## 3.1 Foreign Function Interface

**Report by:** *Manuel Chakravarty*
**Project status:** *Version 1.0 (RC7)*
The Haskell 98 FFI Addendum is meanwhile up to Release Candidate 7 and recently triggered an involved technical discussion on what is the right interface for finalizers for foreign objects. For details, see the archive of the FFI mailing list:
`http://haskell.org/pipermail/ffi/`
After the finalizer debate has settled, there will be another release for public review. The current version of the addendum is available from
`http://www.cse.unsw.edu.au/~chak/haskell/ffi/`
GHC supports the FFI extension as defined in the addendum, in addition to the pre-standard syntax for backward compatibility. Hugs' FFI support has recently been overhauled and mostly brought in line with the FFI Addendum by Alastair Reid. Work on bringing nhc98 closer to the proposed standard is underway.

**Further reading:**

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

## 3.2 Hierarchical Module Namespace

**Report by:** *Simon Marlow*
Activity has shifted towards populating and supporting the new hierarchical libraries (4.1).

## 3.3 Non-sequential Programming

### 3.3.1 Concurrent Haskell

**Report by:** *Simon Marlow*
**Project status:** *maintained, stable*
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/html/base/Control.Concurrent.html`

### 3.3.2 GpH – Glasgow Parallel Haskell

**Report by:** *Phil Trinder*

**The Team:** Phil Trinder, Kevin Hammond, Hans-Wolfgang Loidl, Alvaro Rebon Portillo, Andre Rauber du Bois, Mustafa Aswad, Abyd Al Zain.

**Status:** Steaming Ahead!
GpH aims to provide low-pain parallelism, i.e. acceptable performance with minimal programming effort. It does so by introducing a single new primitive combinator: `par x y` that returns `y` but may create a thread to evaluate `x` depending on machine load. *Evaluation strategies* are higher-order polymorphic functions that abstract over `par` and `seq` to provide high level constructs to coordinate parallelism, e.g. `parList s` applies strategy `s` to every element of a list.
The project has been running since 1994 initially at Glasgow and subsequently at Heriot-Watt and St Andrews Universities. Recent work covers language, system and applications aspects, and consistently emphasises the architecture independence (cf. `http://www.cee.hw.ac.uk/~dsg/gph/arch-indep.html`) of our approach. A robust version of GpH (GUM-4.06) is available for RedHat-based Linux machines (binary snapshot `ftp://ftp.macs.hw.ac.uk/pub/gph/gum-4.06-snap-i386-unknown-linux.tar`; installation instructions `ftp://ftp.macs.hw.ac.uk/pub/gph/README.GUM`). Versions for Sun shared-memory machines, Debian, and an alpha-release based on GHC 5.02 are available on request <gph@cee.hw.ac.uk>.

**Language:** We have recently produced a truly parallel implementation of a referentially transparent bottom-avoiding choice operator ( `http://www.cee.hw.ac.uk/~dsg/gph/papers/drafts/flops-submitted.ps.gz`) and used it to explore a new class of parallel algorithms in GpH, namely branch-and-bound. It reveals an interesting relationship between non-strict and speculative parallel evaluation.

**System:** We are developing the GUM implementation of GpH in the following ways. We are investigating the challenges posed by porting GUM to a computational GRID. To improve the architecture independent performance of GpH

we have added new features to its implementation (GUM). The load balancing ( `http://www.cee.hw.ac.uk/~dsg/gph/papers/drafts/sfp01-gum.ps.gz`) in GUM has been made more flexible by implementing low- and high-watermarks on the spark pools, which represent potential parallelism. Thread migration is being implemented as a technique of avoiding gross load imbalance in applications with a small amount of parallelism. For a better control of *data locality* in GpH programs we are currently exploring language constructs with explicit placement parameters as well as abstractions over these basic constructs. We have improved the distributed shared memory performance (`http://www.cee.hw.ac.uk/~dsg/gph/papers/ps/dsm02.ps.gz`) of GUM, in particular global address management and the graph packing, enabling the user to optimise the parallel execution for execution time or heap space.
An implementation of a time and space static analysis is nearing completion. Although the current analysis is for a strict language, the intention is to use the result of the analysis to select appropriate computations for parallel evaluation.

**Applications:** We have produced detailed comparisons of three parallel functional languages : GpH ( `http://www.cee.hw.ac.uk/~dsg/gph/`), Eden ( `http://www.mathematik.uni-marburg.de/~loogen/eden.html`), PMLS ( `http://www.cee.hw.ac.uk/Research/funct_prog.html`), discussing language and implementation differences ( `http://www.cee.hw.ac.uk/~dsg/gph/papers/drafts/hosc-submitted.ps.gz`). Detailed performance results of several parallel programs on a Beowulf cluster are given. A survey of parallel and distributed Haskells ( `http://www.cee.hw.ac.uk/~dsg/gph/papers/ps/jfp01.ps.gz`) has just appeared in the JFP special issue.
We are investigating the architecture independence of GpH by developing a significant application (genetic alignment) for a variety of parallel architectures: a Beowulf cluster, a Sun-Server SMP. We have also published careful measurements of the Naira parallel Haskell compiler.

**Further reading:**

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

### 3.3.3 Eden

**Report by:** *Rita Loogen and Steffen Priebe*
**Project status:** *updated from GHC 3.x to GHC 5.x*
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 to avoid global garbage collection. The current (freely available) implementation is based on GHC 5.00.2 (beta release). It comprises a library which provides predefined Eden skeletons for many parallel computation patterns like task farms, work-pools, divide-and-conquer etc.
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, optimisations and polytypic extensions.

**Further reading:**

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

## 3.4 Type System/Program Analysis

### 3.4.1 Chameleon/A General Type Class Framework based on Constraint Handling Rules

**Report by:** *Martin Sulzmann*
**Project status:** *on-going*
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. We have incorporated the ideas of the CHR-based overloading approach into an actual programming language called Chameleon. The syntax of Chameleon follows mostly Haskell. We plan to use Chameleon as an experimental test-bed for possible type system extensions.

**Recent developments:**

- Chameleon now comes with a declarative type debugging interface.
- We are currently working on an the Chameleon back-end (i.e. evidence translation)

**Further reading:**

`http://www.comp.nus.edu.sg/~sulzmann/chr/`
`http://www.comp.nus.edu.sg/~sulzmann/chameleon/`

### 3.4.2 Program Analysis for Haskell

**Report by:** *Martin Sulzmann*
**Project status:** *on-going; no significant updates recently*
Our goal is to develop a generic constraint-based program analysis framework for Haskell. We have designed and implemented a binding-time, strictness and exception analysis for Haskell and incorporated both analyses into the GHC compiler. The analysis deals with all features of Haskell such as polymorphic programs and structured data.

**The team:** Kevin Glynn, Harald Sondergaard, Peter Stuckey, Martin Sulzmann

**Further reading:**

`http://www.comp.nus.edu.sg/~sulzmann/mupag/`

## 3.5 Generic Programming

**Report by:** *Johan Jeuring*
Software development often consists of designing a (set of mutually recursive) datatype(s), 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. Another kind of generic function is a function that traverses its argument, and only performs an action at a small part of its argument. 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 (section 5.2.1) is a preprocessor which generates instances of generic functions. It is used in Strafunski (section 4.2.2) to generate a framework for generic programming on terms.

### 3.5.2 Languages

Light-weight generic programming: Generic functions for data type traversals can (almost) be written in Haskell itself, as shown by Ralf Laemmel and Simon Peyton Jones in 'Scrap your boilerplate' (`http://research.microsoft.com/Users/simonpj/papers/hmap/`). In 'Strategic polymorphism requires just two combinators!' (`http://www.cwi.nl/~ralf/ifl02/`), Ralf Laemmel further develops these ideas. Another light-weight approach, using type representations inside Haskell, was presented by Cheney and Hinze at the Haskell workshop.

Generic programs can also be implemented in a language with dependent types, as shown by McBride and Altenkirch in a paper in WCGP'02, see `http://www.dur.ac.uk/c.t.mcbride/generic/`. More about generic programming and type theory ('Generic Haskell in type theory') can be found in Ulf Norells recent MSc thesis `http://www.cs.chalmers.se/~ulfn`.

The Generic Haskell release of last summer supports type-indexed data types, dependencies between generic functions, and special cases for constructors (besides the 'standard' type-indexed functions and kind-indexed types). These extensions are described in the "Type-indexed data types" paper presented at MPC'02, and the "Generic Haskell, Specifically" paper at WCGP'02. The new Generic Haskell release was used in the Summer School in Generic Programming in Oxford last August, at which Ralf Hinze and Johan Jeuring presented two tutorials: Generic Haskell - theory and practice, and Generic Haskell - applications. The former tutorial introduces Generic Haskell, and gives some small examples, the latter paper discusses larger applications such as an XML compressor. More XML tools are described in Paul Hagg's MSc thesis on a framework for developing generic XML tools.

Patrik Jansson adds that there is still a small group at Chalmers working under the slogan "Functional Generic Programming - where type theory meets functional programming" (`http://www.cs.chalmers.se/~patrikj/poly/`), and that while PolyP is not really actively developed anymore, a new version could come out if somebody showed interest.

**Current Hot Topics:** Generic Haskell: implementing the show and read functions from HaXml (section 4.6.1) as generic programs (also for a SchemaToData and XQueryToData tool), an implementation of type checking and inferencing, implementing explicitly recursive generic functions. Other: the relation between generic programming and dependently typed programming; the relation between generic programming and Template Haskell (section 3.6.1).

**Major Goals:** Smaller: Extend Generic Haskell with typechecking. Next release of Generic Haskell: somewhere at the beginning of 2003. Larger: Smoothly integrate generic programming with Haskell programming.

**Further reading:**

`http://repetae.net/john/computer/haskell/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 Meta Programming

### 3.6.1 Template Haskell

**Report by:** *Simon Peyton Jones*

**Team:** *Simon Peyton Jones, Tim Sheard*
Template Haskell is an extension to Haskell that supports *compile-time meta-programming*. The idea is to make it really easy to write Haskell code that executes at compile time, and generates the Haskell program you want to compile.
The ability to generate code at compile time allows the programmer to implement such features as conditional compilation, polytypic programs, macro-like expansion, and the generation of supporting data structures and functions from existing data structures and functions (e.g. the '`deriving`' clause).
Here's a tiny example of conditional compilation:

```
assert :: String -> Expr
assert s | wantAsserts = [| \b v -> if b
                                    then v
                                    else error s |]
         | otherwise   = [| \b v -> v |]
```

To use '`assert`', do this:

```
foo x = $(assert "Uh ho") (x>3) (..foo's rhs..)
```

The `$(...)` construct is a *splice* that says "evaluate this at compile time, and splice in the code returned in place of the splice". The `[| ... |]` is a *quotation* that lets you return a chunk of code as a result.
All this is based on the ideas first put forth by Tim Sheard in MetaML (`http://www.cse.ogi.edu/PacSoft/projects/metaml/`), with the following big difference: Template Haskell is a compile-time-only system, so there is no execution-time cost. As a direct result TH's type system is more generous, and lets you write programs that MetaML would reject.
TH is described in our Haskell Workshop paper (`http://research.microsoft.com/~simonpj/papers/meta-haskell`) and is implemented in GHC. It doesn't appear in any released version yet, but it's in the CVS HEAD and will be in the next release.
Our hope is that by making TH available as part of GHC, people will start to use it for purposes we haven't even dreamt of. Please tell us!

## 3.7 Syntactic Sugar

### 3.7.1 Recursive do notation

**Report by:** *Levent Erkok*
**Project status:** *Implemented in both Hugs and GHC*

**People:** Levent Erkok, John Launchbury
The `do`-notation of Haskell does not allow recursive bindings, that is, the variables bound in a `do`-expression are visible only in the textually following code block. Compare this to a `let`-expression, where bound variables are visible in the entire binding group. It turns out that several applications can benefit from recursive bindings in the `do`-notation, and this extension provides the necessary syntactic support.
If recursive bindings are required for a monadic computation, then the underlying monad should be made an instance of the `MonadFix` class, whose declaration looks like:

```
class Monad m => MonadFix m where
   mfix :: (a -> m a) -> m a
```

The operator `mfix` is required to satisfy several axioms; details can be found on the web-page.
The following instances of MonadFix are automatically provided: `Maybe`, `[]`, `IO`, and `ST` (both lazy and strict).
Jeff Lewis and Sigbjorn Finne helped with the Hugs implementation. GHC implementation was mainly done by Simon Peyton Jones.

**Further reading:**

`http://www.cse.ogi.edu/PacSoft/projects/rmb/`

### 3.7.2 Arrow Notation

**Report by:** *Ross Paterson*
The preprocessor is being maintained, but is now quite stable. It is used by the Yampa (`http://www.haskell.org/yampa/`) system from Yale (section 6.4.3). I'm still taking requests, and am very keen to hear from any users.
`http://www.haskell.org/arrows/`

# Chapter 4

# Libraries

## 4.1 Hierarchical Libraries

**Report by:** *Simon Marlow*

Development continues on the hierarchical libraries, although it has slowed somewhat over the last 6 months. GHC 5.04 shipped this year with the hierarchical libraries, and Hugs is also expected to ship a new version shortly with the hierarchical libraries.

The libraries are now mostly documented with Haddock (section 5.3.5), although we're still missing documentation for many of the "standard" libraries (those that came originally from the Haskell 98 language & library reports). Contributions of more documentation would be gratefully received; it's often just a case of cut-n-paste from the appropriate report adding Haddock syntax as appropriate.

Work on a hierarchical replacement for the aging Posix library is underway, the work in progress can be seen in `fptools/libraries/unix` in the CVS repository. The general plan is to increase portability by using only the FFI (section 3.1) and hsc2hs (the old Posix library used GHC extensions), and to update the library to support functionality from the POSIX 1003.1-2001 standard.

The old hslibs libraries are now almost completely deprecated. GHC itself no longer requires any of them; if you have any code that uses libraries from hslibs (with certain exceptions for those libraries which don't as yet have a hierarchical replacement) then you're encouraged to port your code over to the new libraries - another benefit is that your code will then port to the new release of Hugs much more easily too. For GHC we'll probably do one more major release with hslibs before removing them.

**Further reading:**

`http://www.haskell.org/~simonmar/libraries/libraries.html`
`http://www.haskell.org/~simonmar/lib-hierarchy.html`
`http://www.haskell.org/mailman/listinfo/libraries/`

## 4.2 Data and Control Structures

### 4.2.1 Haskell Foundation Library

**Report by:** *Andrew Bromage*

The intention of the project is to produce a set of foundation libraries suitable for eventual standardisation. At the moment, most of the effort is going into producing and supporting a forked version of Chris Okasaki's Edison library, updated to use post-H98 features.

It's not very complete and only works under GHC at the time of writing. We pretty much have taken over Edison. Or, more correctly, we have forked Edison and are (according to Chris) the only ones actively maintaining an Edison derivative (the interface has changed enough that it's not quite Edison any more).

**Further reading:**

`http://sourceforge.net/projects/hfl/`
For further information, contact:
<hfl-devel@lists.sourceforge.net>

### 4.2.2 Strafunski

**Report by:** *Ralf Lämmel*
**Project status:** *active, maintained*
**Portability:** Hugs, GHC, DrIFT

Strafunski is a Haskell-based bundle for generic programming with functional strategies, that is, generic functions that can traverse into terms of any type while mixing type-specific and uniform behaviour. This style is particularly useful in the implementation of program analyses and transformations.

Strafunski bundles a Haskell library StrategyLib and some tools along with it, most notably a precompiler for user-supplied datatypes. The StrategyLib provides themes such as simple traversal schemes and generic algorithms for name analysis. The precompiler is simply a customised version of DrIFT (section 5.2.1).

The Strafunski-style of generic programming can be seen as a lightweight variant of generic programming (section 3.5) Pros: simplicity (no language extension, generic functionality just relies on a few original combinators), domain support (program transformation and analysis is addressed by the library and various examples). Cons: restrictions (focus on term traversal). Generic Haskell (section 3.5) provides a more powerful language for polytypic programming, think of anamorphisms, functorial maps, and type-indexed dataypes.

**Further reading:**

`http://www.cs.vu.nl/Strafunski/`

## 4.3 Graphical User Interfaces

### 4.3.1 HTk

**Report by:** *Christoph Lüth and George Russell*
**Project status:** *beta release*
HTk is an encapsulation of the graphical user interface toolkit and library Tcl/Tk for the functional programming language Haskell. It allows the creation of high-quality graphical user interfaces within Haskell in a typed, abstract, portable manner. HTk is known to run under Linux, Solaris, Windows 98, Windows 2k, and will probably run under many other POSIX systems as well. HTk works with GHC, versions 5.02.3 and later.

**Further reading:**

`http://www.informatik.uni-bremen.de/htk`

### 4.3.2 Object I/O for Haskell

**Report by:** *Krasimir Angelov*
**Project status:** *completed, maintained; port needed*
The Object I/O is a flexible library for building rich user interfaces. It is a port of the popular Clean Object I/O to Haskell (`http://www.cs.kun.nl/~clean/`). The current implementation for Clean and Haskell supports only the Windows platform but the library is done keeping in mind its portability. The aim is to create a highly portable GUI library, so that programs will be translated to different platforms without rewriting while giving the programs a native look and feel for the target platform. The main difference between Object I/O and TclHaskell, FranTk and some other, is that Object I/O uses a native interface (Win32 API for Windows and GTK+ for Linux) instead of a scriptable interface (Tcl/Tk). This is more difficult to implement but is more effective.
The library uses nonstandard type system extensions: explicit universal quantification and existentially quantified data constructors. The current implementation works only with GHC-5.02 or higher compatibles and has moved from the hslibs collection to the new libraries. The package is distributed together with the port of original examples contributed with Clean Object I/O. These examples help the customers understand how to work with the library and how to understand the differences between the implementation for Haskell and Clean (for these who have experience with Clean). There is also a draft of Object I/O quick reference.
The library is functionally completed, but the attempt to port the library to Gtk (as the basis for a Linux version) turned out too difficult and is still unfinished. Also no attempt has been made to build the library with nhc or hugs. Krasimir is still available for bug fixes, but apart from those, he has now moved on to other projects, building on Object I/O, but also trying some new ideas about GUI library design (more about those in the next edition;-). He would be happy to assist, if someone else would come forward to complete the Gtk port.

**Further reading:**

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

### 4.3.3 Gtk+HS

**Report by:** *Manuel Chakravarty*
**Project status:** *beta release*
Gtk+HS is a Haskell binding to the GTK+ GUI toolkit `http://www.gtk.org/`, which is the toolkit on which the Gnome desktop is based. GTK+ is a fully-fledged modern widget set and all its basic and some of its advanced functionality is already available from Haskell. The current binding is to GTK+ 1.2. Support for the new GTK+ 2.0 API is the next item on the todo list.
The binding is currently at version 0.14.10, which is a full binary release published in September 2002. More details as well as source and binaries packages are at

**Further reading:**

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

### 4.3.4 Gtk2hs

**Report by:** *Axel Simon*
**Project status:** *beta*
Gtk2hs is a wrapper around the latest Gtk release (Version 2.0 or Gtk 2 for short). Although it provides a similar low level veneer like Gtk+HS (section 4.3.3), it is completely rewritten from scratch, circumventing some of the problems the Gtk+HS has:

- To retain backwards compatibility, Gtk+HS does not make use of ForeignPtr which moves the burden of memory management onto to the application developer. This defeats the benefits of the automatic memory management of Haskell and makes Gtk+HS unsuitable for larger applications. The interface of Gtk2hs ensures that all objects and structures are reference-counted and freed properly. The implementation became much easier since Gtk 2 turned most structures into GObjects (a root class like Java's Object) which allows us to use the same well-tested infrastructure for memory management. The downside is that our library only supports GHC 5.02 and higher as it is the only compiler implementing the FFI (section 3.1) to the extent that we need.
- While signals (call-backs from Gtk to the user application) in Gtk+HS are handled in an ad-hoc fashion for each widget, Gtk2hs uses a list of prototypes which ships with Gtk+ to automatically generate appropriate glue code. This code makes it possible to bind a signal with just one line. As a result, Gtk2hs provides access to many more signals than Gtk+HS.
- The type of a widget is expressed in the same way as it is in Gtk+HS. But while the class and instance definitions are written by hand in Gtk+HS, we use a simple text file depicting the hierarchy of the widgets (which is included with the Gtk documentation) and generate the definitions automatically. This enabled us in the past to quickly change names (e.g. we got rid of the Gdk namespace) and to add casting functions.

- Only the new version of Gtk has actually a port to the Windows platform.

Beyond these technical differences, Gtk2hs strives for the following goals:

- Provide a complete binding to the Gtk library. We estimate that 80-90% of the functionality is already available.
- Provide the first portable binding to a single GUI toolkit. We know there is ghc and Gtk2 for Win32 platforms, but we haven't tried yet.
- Provide a well-documented library. A first attempt of documentation can be viewed at `http://www.cs.ukc.ac.uk/people/staff/as49/gtk2hs`
- A thin convenience wrapper called Mogul provides useful abstractions that go beyond the pure binding. Specifically it allows users to look up widgets by name (like libglade).
- We have a small tool in preparation which generates Haskell code from the output of the Glade GUI builder. This approach seems to be much more usable than a binding to libglade.

Our current work is done in a CVS repository which can be found on `http://sourceforge.net/projects/gtk2hs/`. We plan to finish the library by Summer next year.

## 4.4 Graphics

### 4.4.1 HGL Graphics Library

**Report by:** *Alastair Reid*
**Project status:** *Maintained, stable*

The HGL gives the programmer access to the most interesting parts of the Win32 and X11 library without exposing the programmer to the pain and anguish usually associated with using these interfaces. The library is distributed as open source and is suitable for use in teaching and in applications. The library currently supports:

- Filled and unfilled 2-dimensional objects (text, lines, polygons, ellipses); bitmaps (Win32 version only, for now); control over text alignment, fonts, color.
- Simple input events (keyboard, mouse, window resize) to support reactivity; timers and double-buffering to support simple animation.
- Use of concurrency to avoid the usual inversion of the code associated with event-loop programming.
- Multiple windows may be handled at one time.

To keep the library simple and portable, the library makes no attempt to support:

- User interface widgets (menus, toolbars, dialog boxes, etc.); palette manipulation and other advanced features.
- Many kinds of input event.

**Status:** The library works on both Win32 and X11 under Hugs and (unsupported) GHC. The API is stable and the library is used throughout Paul Hudak's 'School of Expression' textbook (`http://haskell.org/soe/`). The last release was 2.0.4 in December 2001. A release that works better with the new release of Hugs (notably the support for hierarchial module namespace) and current releases of GHC will be available soon.

**Further reading:**

HGL web page: `http://haskell.org/graphics/`
School of Expression web page: `http://haskell.org/soe/`
Author's web page:
`http://www.reid-consulting-uk.ltd.uk/alastair/`

### 4.4.2 FunGEn – A game engine for Haskell

**Report by:** *Andre W B Furtado*
**Project status:** *being rebuilt*

The objective of the FunGEn project is to create a high-level game engine in and for Haskell. A game engine, roughly speaking, is a tool intended to help a game programmer to develop games in a faster and automated way, avoiding him to worry about low-level implementation details. The main advantage of using a game engine is that, if it is built in a general and modular architecture, it can be used to develop many different (types of) games.
The first release of FunGEn (April/2002) consisted of a 2D platform-independent game engine, which implementation was based in HOpenGL (Haskell Open Graphics Library). It supported:

- Initialization, updating, removing, rendering and grouping routines for game objects; definition of a game background (or map), including texture-based maps and tile maps; loading and displaying of 24-bit bitmap files;
- Reading and interpretation of the player's keyboard input; collision detection; time-based functions and predefined game actions; a few debugging and game performance evaluation facilities;
- Sound support (for windows platforms only... :-[ )

Some feedback indicated that the first version of FunGEn was not as "functional" as it was desired: some game issues were still being dealt through an imperative fashion. This way, the authors of this project decided to change the game engine philosophy: programmers should describe a game as a set of "specifications" rather than defining its behavior imperatively. One plausible alternative for accomplishing this task is porting the Clean Game Library (CGL) to Haskell, adding some FunGEn specific features. Hence, this is the actual status of the FunGEn project: it is being rebuilt in order to provide game programming mechanisms following the CGL concepts. This really demands some time, but the authors expect a new version to be released soon.

The final objective of FunGEn is to support both 2D and 3D enviroments, some game programming tools (such as map editors) and advanced game functionalities (such as multiplayer networking), although it is actually far away from that.

FunGEn is being maintained at the Informatics Center of the Federal University of Pernambuco, by Andre W B Furtado (assisted by lecturer Andre Santos), and it's wide open for any implementation contributions. We would like to thank Mike Wiering, the creator of Clean Game Library.

**Further reading:**

`http://www.cin.ufpe.br/~haskell/fungen/`
`http://www.cin.ufpe.br/~haskell/hopengl/`
`http://www.haskell.org/HOpenGL/`

### 4.4.3 FunWorlds – Functional Programming and Virtual Worlds

**Report by:** *Claus Reinke*
**Project status:** *being rebuilt*

FunWorlds is an ongoing experiment to investigate language design issues at the borderlines between concurrent systems, animated interactive 2&3d graphics, and functional programming. One of the aims is to get a suitable platform for expressing such things, preferably from Haskell.

Our earlier prototypes translated scene descriptions from a Haskell-embedded DSL combining ideas from Fran and VRML into standard VRML+ECMAScript (reported at IFL'2001), but due to some cumbersome VRML restrictions, this is currently being reimplemented. The new prototypes are built on Sven Panne's HOpenGL Haskell binding to OpenGL. This means that it is no longer easy to import ready-made high-level functionality from VRML-browsers, but we've got access to functional programming concepts at runtime, not just at compile time.

The focus so far, as reported at the recent IFL'2002, has been on a redesign of some fundamental Fran concepts, towards a simpler operational semantics as a basis for uncomplicated implementations with more predictable performance characteristics. At the same time, we don't want to throw out too much of Fran's high-level modeling approach.

An initial release is planned for later this year; this will not have substantially more functionality than the snapshot used for IFL'2002, so it will be pretty basic. The main obstacle on the way is to write some form of introductory tutorial on the new DSEL design and how one might use it. Once we've got some more experience with the language basics, graphics functionality will be added on demand.

**Further reading:**

`http://www.cs.ukc.ac.uk/people/staff/cr3/FunWorlds/`

## 4.5 Tool Frameworks

Instead of developing fixed tools, it is sometimes possible to generalize the code implementing the tool functionality into a library, so that the code can be reused for a family of tools.

### 4.5.1 Medina – Metrics for Haskell

**Report by:** *Chris Ryder*

The Medina library is a Haskell library for GHC that provides tools and abstractions with which to build software metrics for Haskell. The library includes a parser and several abstract representations of the parse trees, some visualisation systems including pretty printers and HTML generation, and now includes some integration with CVS to allow temporal operations such as measuring a metric value over time. This is linked with some simple visualisation mechanisms to allow exploring such data. Recently support for generating call graphs of programs has been added, including a visualisation system to browse such call graphs. A case study has been started to work towards some validation of metrics by looking at the change history of a program and how various metric values evolve with those changes.

The Medina project collaborates with the Refactoring project (section 5.3.3), also at UKC.

**Further reading:**

`http://www.cs.ukc.ac.uk/people/rpg/cr24/medina/`

## 4.6 Web Programming

### 4.6.1 HaXml

**Report by:** *Malcolm Wallace*
**Project status:** *stable, maintained*

The HaXml project is still alive, in stable maintenance mode, now at version 1.07b. HaXml provides many facilities for using XML from Haskell. The most user-visible change recently has been to convert the HaXml libraries to the new hierarchical namespace. We have also recently provided a validator for checking documents against a DTD.

**Further reading:**

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

### 4.6.2 HXml

**Report by:** *Joe English*
**Project status:** *pre-beta, version 0.2*

HXML is a non-validating XML parser written in Haskell. It is designed for space-efficiency, taking advantage of lazy evaluation to reduce memory requirements. HXML may be used as a drop-in replacement for the HaXml (section 4.6.1) parser in existing programs. HXML includes a module with functionality similar to HaXml's 'Combinator' module, but recast in an Arrow-based (section 3.7.2) framework.

HXML also provides multiple representations for XML documents: a simple algebraic data type containing only the essentials (elements, attributes, and text), a tree representation which exposes most of the full XML Information Set, and a navigable tree representation supporting all of the principal XPath axes (ancestors, following-siblings, etc).
HXML has been tested with GHC 5.02, GHC 5.04, NHC 1.12, and most recent versions of Hugs. NHC 1.10 requires a patch. HXML is basically in maintenance mode right now until I can find some spare time; support for XML Namespaces is next on the TODO list.

**Further reading:**

`http://www.flightlab.com/~joe/hxml/`

### 4.6.3 Haskell XML Toolbox

**Report by:** *Uwe Schmidt (uwe@fh-wedel.de)*
**Project status:** *first release*
The Haskell XML Toolbox is a collection of tools for processing XML with Haskell. It is itself purely written in Haskell. The core component of the Haskell XML Toolbox is a validating XML-Parser that supports almost fully the Extensible Markup Language (XML) 1.0 (Second Edition).
The Haskell XML Toolbox bases on the ideas of HaXml (section 4.6.1) and HXML (section 4.6.2), but introduces a more general approach for processing XML with Haskell. The Haskell XML Toolbox uses a generic data model for representing XML documents, including the DTD subset and the document subset, in Haskell. This data model makes it possible to use filter functions as a uniform design of XML processing applications. The whole XML parser including the validator parts was implemented using this design. Libraries with filters and combinators are provided for processing the generic data model.

**Features:**

- validating XML parser
- full Unicode support
- uniform data modell for DTDs and XML content
- support of http: and file: protocol
- tested with W3C XML validation suite

**Current Work:**

- XPath implementation
- XSLT implementation
- better error reporting

**Further reading:**

The Haskell XML Toolbox Webpage `http://www.fh-wedel.de/~si/HXmlToolbox/index.html` includes downloads, online documentation and a master thesis describing the design of the toolbox.

### 4.6.4 WASH/CGI – Web Authoring System for Haskell

**Report by:** *Peter Thiemann*
WASH/CGI is an embedded DSL (read: a Haskell library) for server-side Web scripting based on the purely functional programming language Haskell. Its implementation is based on the portable common gateway interface (CGI) supported by virtually all Web servers. WASH/CGI offers a unique and fully-typed approach to Web scripting. It offers the following features

- a monadic interface to generating HTML output
- type-safe compositional approach to specifying form elements; callback-style programming interface for forms
- automatic error detection
- complete interactive script in one program
- type-safe interfaces to state with different scopes: interaction, persistent client-side (cookie-style), persistent server-side
- integration with CSS yields compositional style descriptions
- on-the-fly generated graphics
- high-level interface to email generation

**Current work** includes

- new version with Haddock-generated documentation is upcoming; in consequence, some things have been streamlined and some obsolete entry points have been removed
- support for hooking directly into Simon Marlow's Haskell Webserver (so that I can finally get started on implementing authentication)
- support for generating forms that can be stored in files or sent via email or news. Anyone picking up the form can resume the interaction.
- probably support for frames in some form
- beyond-Haskell98 interface: some additional guarantees can be gained by using multi-parameter type classes and rank-2 polymorphism in two places.

Items still on the to do list

- incorporation of WASH/HTML, a typed interface for generating mostly valid HTML documents
- preprocessor for translating markup in XML syntax into WASH/HTML
- database interface
- authentication
- user manual

**Further reading:**

WASH Webpage `http://www.informatik.uni-freiburg.de/~thiemann/WASH/` includes examples, a tutorial, papers about the implementation.

# 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 latest binary release, version 0.10.17, was published in September 2002. The tool significantly simplifies binding development by automatically translating C types and enums into Haskell data types, managing access to C structs, and semi-automatically generating marshalling code for arguments and results of C functions. It has been stress tested in the development of the Gtk+HS GUI library (section 4.3.3). Source and binary packages as well as a reference manual are available from

**Further reading:**

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

### 5.1.2 GreenCard

**Report by:** *Alastair Reid*
**Project status:** *Maintained, stable*
**Portability:** *Hugs, GHC, NHC and C, C++*

GreenCard is a foreign function interface preprocessor for Haskell and has been used (amongst other things) for the Win32 and X11 bindings used by Hugs and GHC. Source and binary releases (Win32 and Linux) are available. The last release was 2.0.4 (August 2002). A release that provides access to and takes advantage of the new Foreign Function Interface libraries will be available soon.

**Further reading:**

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

### 5.1.3 Java VM Bridge

**Report by:** *Ashley Yakeley*

Java VM Bridge is a GHC package intended to allow full access to the Java Virtual Machine from Haskell, as a simple way of providing a wide range of imperative functionality. Its big advantage over earlier attempts at this is that it includes a straightforward way of creating Java classes at run-time that have Haskell methods (using DefineClass and the Java Class File Format). It also features reconciliation of thread models without requiring GPH.
It is intended to make writing "Java in Haskell" as straightforward as possible. To this end, each Java class is a separate type, and the argument lists of methods of automatically-generated interfaces to Java classes make use of subtype class relations to minimise explicit upward casting. Java exceptions are represented as Haskell monadic exceptions, and may be caught or thrown accordingly. Also, the two garbage collectors are integrated in such a way that cross-collector reference loops won't happen.
As a point of cleanliness and principle, it makes no use of "unsafe" Haskell calls (or pure function FFI). The layered design allows access to either lifted monads that keep track of context data (specifically, the JNIEnv pointer) and do all the work of preloading for you, or "IO"-based functions if you want to do all that yourself.

**Current Status:** A beta-quality 0.1 was released in December 2001, for x86 Unix only. Release 0.2 will also be available for Windows and MacOS X, sometime in November.

**Contact:** Ashley Yakeley <ashley@semantic.org>

**Further reading:**

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

## 5.2 Meta Programming

*"Why write a program when you can write a program to write a program?"* (author unknown).
Even in a language where functions are first-class citizens, you sometimes want to write programs at a meta level, be it to get that extra leverage in productivity, to test some ideas for language extensions, for debugging/instrumenting your code, or for analyses and transformations. Unfortunately, generic tool support for this kind of tasks has been somewhat lacking, so that Haskell meta-programmers currently have to implement their tools almost from scratch (Drift, HAT, Sugar for Arrows, Haddock, labelled fields before they became part of the language, ...).

Not only is this a sub-optimal use of precious developer time, it also carries the risk of tools representing substantial investments being left behind as their developers move on to other commitments. The latest example is Hatchet, a Haskell-in-Haskell frontend incorporating not only parsing and pretty-printing, but also a type system. Introduced only in our previous edition, it is now looking for a new home (please contact Bernie Pope if you are interested).

In this section, we hope to document any progress being made in this area, but see also the language extensions chapter for Template Haskell (section 3.6.1).

### 5.2.1 Haskell Preprocessors

**DrIFT**

**Report by:** *John Meacham*

DrIFT is a type sensitive preprocessor for Haskell. It extracts type declarations and directives from modules. The directives cause rules to be fired on the parsed type declarations, generating new code which is then appended to the bottom of the input file. The rules are expressed as Haskell code, and it is intended that the user can add new rules as required.

DrIFT automates instance derivation for classes that aren't supported by the standard compilers. In addition, instances can be produced in separate modules to that containing the type declaration. This allows instances to be derived for a type after the original module has been compiled. As a bonus, simple utility functions can also be produced from a type.

DrIFT is very close to the big 2.0 release once I fix a couple known bugs.

**Further reading:**

`http://repetae.net/john/computer/haskell/DrIFT/`

### 5.2.2 Scanning, Parsing, and Analysis

See also constraint-based program analysis (section 3.4.2), and the group of tools developed in Utrecht (section 6.4.5).

**Happy**

**Report by:** *Simon Marlow*
**Project status:** *stable, maintained*

Happy has seen one new release, 1.13, in June of this year, which was mainly a bugfix release.

On the development front, there's nothing major to report, except that the changes for GLR parsing are still waiting in the wings and I'm talking to the author of Alex (Chris Dornan) about a possible merge of Happy & Alex at some point.

**Further reading:**

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

### 5.2.3 Haskell Transformations

**Report by:** *Eelco Visser*

**HSX** The HSX framework is a prototype for experimentation with the application of rewriting strategies using the transformation language Stratego to program optimization. Although the syntax is for complete Haskell (without layout), the transformations are done on a core-like subset only. The framework was used to implement the Warm Fusion transformation for deforestation that turns recursive function definitions into build/cata form. This form makes deforestation, the fusion of a composition of data structure producing and consuming functions, a piece of cake. Extension of the work to full Haskell was not continued by lack of a reusable Haskell front-end.

**HsOpt** Recently we have started to work on a new version of the framework called HsOpt. This is a transformation framework for Helium, a proper (light) subset of Haskell developed at Utrecht University (section 6.4.5). We reuse the parser, prettyprinter, and typechecker from the Helium project. The first target is the specification of a GHC style simplifier in Stratego. The Haskell ATerm library is used to interface Haskell components and Stratego components.

**Further reading:**

`http://www.stratego-language.org/Stratego/HSX`
`http://www.stratego-language.org/Stratego/HsOpt`

## 5.3 Program Development

### 5.3.1 Tracing and Debugging

**Report by:** *Olaf Chitil and Bernie Pope*

There are a number of tools with rather different approaches to tracing Haskell programs for the purpose of debugging and program comprehension. In particular Hood and Hat seem to become increasingly popular.

Both **Hood**, the portable library for observing data structures at given program points, and **GHood**, the graphical variant for animated observations have remain unchanged for over a year.

On 14 June version 2.00 of **Hat**, the Haskell tracing (and debugging) system, was released. It is distributed separate from nhc98 and can be used both with nhc98 and ghc. The compiled program generates a trace file alongside its computation. With several improved tools the trace can be viewed in various ways: algorithmic debugging a la Freja; Hood-style observation of top-level functions; backwards exploration of a computation, starting from (part of) a faulty output or an error message. All tools inter-operate and use a similar command syntax. A tutorial explains how to generate traces, how to explore them, and how they help to debug Haskell programs. Hat 2.00 requires programs to strictly conform to the Haskell 98 standard. A new release that supports hierarchical modules, more libraries, multi-parameter classes with

functional dependencies and improved performance with ghc will appear by the end of the year.

**Buddha** is a declarative debugger for Haskell 98. Each module in the program undergoes a transformation to produce a new module (as Haskell source). The transformed modules are compiled and linked with a library for the interface, and the resulting program is executed. The transformation is crafted such that execution of the transformed program constitutes evaluation of the original (untransformed) program, plus construction of a semantics for that evaluation. The semantics that it produces is a tree with nodes that correspond to function applications.

Currently buddha works with GHC version 5.04 or greater. No changes to the compiler are needed. There are no plans to port it to other Haskell implementations, though there are no significant reasons why this could not be done.

Version 0.1 of buddha is freely available as source. This version supports most of Haskell 98, however there are a few small items that are not supported. These are listed in the documentation. Future releases will include support for the missing features, and a much improved user interface.

At IFL 2002, Frank Huch and Thomas Boettcher presented a debugger for Concurrent Haskell (section 3.3.1). It has a graphical user interface for visualising the state of threads and communication abstractions. It is based on replacing the Concurrent library by a library ConcurrentDebug with the same interface. A first public release will appear soon at `http://www.informatik.uni-kiel.de/~fhu/chd/`

Also at IFL 2002, Alcino Cunha, Jose Barros and Joao Saraiva presented the prototype of a tool for graphically animating the evaluation of recursive Haskell functions.

**Further reading:**

`http://www.haskell.org/libraries/#tracing`
`http://www.cs.mu.oz.au/~bjpop/buddha`

### 5.3.2 Development Environments

**hIDE**

**Report by:** *Duncan Coutts*
**Project status:** *being rewritten*

hIDE is an integrated development environment for Haskell (primary author: Jonas Svensson).

In the last 6 months, hIDE in its current form has been frozen. A complete rewrite is now underway which aims to provide an attractive gtk2 interface and a plugin system. For people reading this report, the plugin system is probably the most important new feature.

The significance of this is that you will be able to use a Haskell IDE with Haskell as its extension language. No longer will you have to delve into emacs' elisp or another foreign languages to get your favourite tool to work with your editor. Ultimately, we would love to see plugins for many of the excelent tools listed in this report.

Development has been a little slow recently as we work out the basic architecture. Help would be welcomed. We will announce a development release when the core is finished and the API for plugins is more stable.

**Further reading:**

`http://www.dtek.chalmers.se/~d99josve/hide/`
`http://sourceforge.net/projects/haide`
`haide-devel@lists.sourceforge.net`

### 5.3.3 Refactoring

**Report by:** *Claus Reinke*
**Team:** *Simon Thompson, Huiqing Li, Claus Reinke*

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. Like most Haskellers, we had developed a habit of refactoring our programs on a small scale long before we learned that others would call this refactoring. And in a functional language like Haskell, we are much more adventurous about the scale of program transformations we will try to improve our code. Without proper tool support, however, this soon gets out of hand, and only backups and undo come to the rescue of the over-zealous code-improver.

We want to explore the wealth of functional program transformation research to bring refactoring to Haskell programmers. Our project finally got underway this July, and apart from the usual background work – (re-)reading existing work on refactoring catalogues and tools, surveying Haskellers' editing habits, and doing some casestudies – we have now started to experiment with Haskell frontends (parsing, pretty-printing, type-checking) and other tools (strategic programming) that could form the foundations for actually implementing a refactoring tool for Haskell.

Tools we've looked at so far include HsParser, Haddock's parser (section 5.3.5), Hatchet (section 3.6), parts of Programmatica, and Strafunski (section 4.2.2). While all of these are helpful in different ways, they are also limited in different ways – chosing some of them and making them work together on real-world Haskell sources to the extent required for our project has proven to be challenging so far. This is partially due to the tools being relatively new, partially due to each of them focussing only on a specific job that might not match our more general requirements.

The issues relating to infrastructure for Haskell meta-programming (section 3.6) require, and would merit, more attention and further development. Because of the lack of a standard interface to Haskell frontend information (compare, e.g., ADA's Semantic Interface Specification `http://www.acm.org/sigada/wg/asiswg/`), Haskell tool and IDE developers keep reinventing wheels and unsatisfactory hacks. We are also looking into XML/XSLT support for organising and maintaining our database of functional refactorings, and we collaborate with the Medina project (section 4.5.1).

**Further reading:**

`http://www.cs.ukc.ac.uk/research/groups/tcs/fp/Refactor/`

### 5.3.4 Testing

#### HUnit

**Report by:** *Dean Herington*
**Project status:** *maintained, no major changes*

Hunit is a unit testing framework for Haskell similar to JUnit for Java. With HUnit, a Haskell programmer can easily create tests, name them, group them into suites, and execute them, with the framework checking the results automatically. Test specification is concise, flexible, and convenient.

A minor revision (HUnit 1.1) is in the works and should be out soon (1-2 weeks). Its main purpose is to adapt to recent GHC (5.04) and Hugs (Oct. 2002) versions.

HUnit is free software that is written in Haskell 98 and runs on Haskell 98 systems. The software and documentation can be obtained at `http://hunit.sourceforge.net`.

#### QuickCheck

**Report by:** *John Hughes*
**Project status:** *maintained, new features*

QuickCheck is a tool for testing Haskell programs automatically. The programmer provides a specification of the program, in the form of properties which functions should satisfy, and QuickCheck then tests that the properties hold in a large number of randomly generated cases. Specifications are expressed in Haskell, using combinators defined in the QuickCheck library. QuickCheck provides combinators to define properties, observe the distribution of test data, and define test data generators.

We have been using QuickCheck to test monadic code, especially in the ST monad, and there is a new version available with combinators for defining "monadic properties". Our Haskell Workshop paper explains how to use them.

We are also experimenting with combinators to make test data generators easier to write, with using Generic Haskell (section 3.5) for the same purpose, and with integration with the Hat tracer. Some of this was presented at the Advanced Functional Programming summer school this year.

We've been planning for a while to combine all our experimental versions into one stable new version, but teaching has got in the way.

**Further reading:**

`http://www.cs.chalmers.se/~rjmh/QuickCheck/`

### 5.3.5 Documentation

#### Haddock

**Report by:** *Simon Marlow*

Haddock saw many improvements in features and stability culminating in the 0.4 release at the end of July this year. This release of Haddock is used to generate the documentation for the hierarchical libraries distributed with recent releases of GHC.

Haddock is now described in a paper, which appeared at the Haskell Workshop 2002:
`http://www.haskell.org/~simonmar/papers/haddock.ps.gz`

At this point I consider Haddock to have most of the main feature set I had originally intended. There is however still a long list of things to do: see the file TODO in the Haddock source tree, and comments/contributions are of course always welcome.

**Further reading:**

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

# Chapter 6

# Applications, Groups, and Individuals

## 6.1 Non-Commercial Applications

This section lists applications developed in Haskell, be it in academia, in industry, or just for fun, which achieve some non-Haskell-related end.

### 6.1.1 HScheme

**Report by:** *Ashley Yakeley*

HScheme is a Scheme interpreter written in Haskell. There's a stand-alone interpreter program, or you can attach the library to your program to provide "Scheme services". It's very flexible and general with types, and you can pick the "monad" and "location" types to provide such things as a purely functional Scheme, or a continuation-passing Scheme (that allows call-with-current-continuation), or a fixed-point Scheme (that allows call-with-result), etc.

Current status: It's close to R5RS, but it's currently missing certain functionality such as inexact numbers and vectors. There have been no releases: you'll have to download and build from CVS if you want to use it. But you can play with the interpreter on the web at `http://hscheme.sourceforge.net/interpret.html`.

It's not particularly fast.

**Further reading:**

`http://hscheme.sourceforge.net/`

Contact: <ashley@semantic.org>

### 6.1.2 Hume: a Language for Embedded Real-Time Systems

**Report by:** *Kevin Hammond*

**Project status:** *ongoing*

Hume is a strict functionally-based concurrent language incorporating high-level coordination constructs. The language is aimed at hard real-time, low memory (e.g. embedded systems) settings. The expression layer is purely functional, with a syntax that is similar to that of Haskell.

Hume programs are short and have a small footprint (< 64K including RTS and heap is possible on RTLinux). Compared with Java KVM (which is aimed at embedded systems), Hume programs are 10x faster, and require about 50% of the memory. It is possible to statically cost memory requirements for a restricted (but usable) subset of the language. We are working on improving this using a novel type-and-effect system. We have already demonstrated hard real-time capability on RTLinux and are now working towards a complete embedded systems implementation on e.g. Lego Mindstorms robots, or semi-autonomous vehicles.

The front-end tools (lexer, parser, cost modeller etc.) are all Haskell-based, using Alex and Happy (section 5.2.2) as appropriate. There is a reference interpreter written in Haskell and an abstract machine (bytecode) compiler/interpreter. The compiler is written in Haskell, with the bytecode interpreter written in C for portability and performance. We are now working on improved code generators, using e.g. threaded code. These tools should be seen as research-quality! We have been impressed by Haskell's ease/speed of construction/modification, conciseness and robustness (this is an interesting experience for a language designer/implementor!).

**Further reading:**

`http://www.hume-lang.org`

### 6.1.3 ParaGAP: Parallel Symbolic Computer Algebra

**Report by:** *Kevin Hammond*

**Project status:** *new project*

This project aims to provide parallel programming support for the GAP computer algebra system on a range of parallel architectures. This will be achieved by calling GpH library functions (section 3.3.2) from within GAP code. We anticipate two useful outcomes for the Haskell community:

1. libraries of functions to manage groups and permutations using the latest computer algebra technology;
2. a new user base.

**Further reading:**

`http://www.dcs.st-and.ac.uk/~kh/paraGAP.pdf`

### 6.1.4 Knit

**Report by:** *Alastair Reid*
**Project status:** *Active, maintained, no recent news*
**Portability:** *GHC (maybe Hugs, still), Linux, FreeBSD*

Knit is a component definition and linking language for systems programming based on the Unit component programming model. Knit lets you turn ordinary C code (e.g., bits of the Linux kernel) into components and link them together to build new programs. Since the freedom to do new things brings with it the freedom to make new errors, Knit provides a simple constraint system to catch component configuration errors. Knit also provides a cross-component inliner and schedules initialization and finalization of components.

Knit is released under a BSD-style license, is written in Haskell (and a little C) and includes a C parser and pretty-printer. A useful little utility included in the distribution is a tool for renaming symbols in ELF-format object files.

Current work aims to extend error checking into the real-time domain, to automate generation of components, and to turn Knit into an architecture description language (ADL) instead of just a module interconnection language (MIL).

**Further reading:**

`http://www.cs.utah.edu/flux/alchemy/`

## 6.2 Commercial Applications

### 6.2.1 Reid Consulting Ltd

**Report by:** *Alastair Reid*

Many companies are starting to allow their programmers to develop small prototypes in Haskell but few are willing to take a chance on using Haskell on a large project. The risks to these companies include lack of support for tools, lack of tutorials and training courses, gaps in the set of available libraries, and lack of 'gurus' to call on when things go wrong.

Reid Consulting can meet those needs. Our background and continuing involvement in the development of Haskell tools and compilers (GreenCard, Hugs, GHC, etc.) and the Haskell language and library design (the Haskell report, the Standard libraries, the Hugs-GHC libraries, the Foreign Function Interface and the HGL Graphics Library) and our use of Haskell to develop large systems, provide the experience and the contacts needed for effective support of real projects.

Where acceptable to clients, we have a policy of releasing any fixes or developed code as OpenSource for use by the wider Haskell community.

**Further reading:**

For more information, see `http://www.reid-consulting-uk.ltd.uk/` or contact Alastair Reid <alastair@reid-consulting-uk.ltd.uk>

### 6.2.2 Binary Parser

**Report by:** *Sengan Baring-Gould*

Sengan Baring-Gould <Sengan.Baring-Gould@nsc.com> at National Semiconductor is developing a binary parser which given a grammar is able to extract fields from values. This is used as part of an internal ICE (hardware debugger).
The declarations follow the format:

```
declaration :: supertype
            = { fields in order, msb first,
                allowing bitslicing }
            : { local fields if any, and
                their types or lengths };
```

For instance the definition of the eax register on a x86 is:

```
eax = { ex, ax } : { ex[16] };
ax  = { ah, al };
```

The grammar provides types that can be arranged into a hierarchy so that fields whose location moves depending on bits within the value can be automatically found, and referenced. For instance `cs` is a descriptor and there are 20 kinds of descriptors, two of which are illustrated below.

```
cs :: Descriptor32;

Small_Code_Segment
  :: Descriptor32
   = { base[31:24], 1'b0, is_32_bit, 1'bx,
       available, limit[19:16], present, dpl,
       2'b11, conforming, readable, a,
       base[23:0], limit[15:0] }
   : { is_32_bit[1], present[1], conforming[1],
       readable[1], a[1], dpl[2], available[1],
       base[32], LimitBytes limit };

Small_Data_Segment
  :: Descriptor32
   = { base[31:24], 1'b0, is_32_bit, 1'bx,
       available, limit[19:16], present, dpl,
       2'b10, expand_down, writable, a,
       base[23:0], limit[15:0] }
   : { is_32_bit[1], present[1], expand_down[1],
       writable[1], a[1], dpl[2], available[1],
       base[32], LimitBytes limit };
```

In all cases `cs.dpl` will access the relevant field even if it were in different places.

Binary parser provides the ability to reference by name values which may be composed of other values. It goes one step further in that the client program does not need to know where particular value is buried, only what its value is. Binary parser grammars are intended to enable non-programmers to access fields of their registers, without requiring the ICE-developer to write explicit code to do so. For instance a technical writer could write a binary parser grammar for a device of which the ICE developer has never heard. Stress

has been put on generality and simplicity, rather than efficiency. For instance binary parser allows multiple definitions, cyclic definitions, etc.
Binary Parser is implemented in Haskell whereas the current ICE is not (C++) – but the next generation will be. Currently communication is achieved using pipes so as to be compatible with both windows and unix (binary parser is used by 2 internal tools, one is unix one is windows).
Binary parser simplifies the porting of the ICE from chip to chip where the location of register-fields may change but their functionality does not.

### 6.2.3 Extending Lava for System on Chip Designs

**Report by:** *Satnam Singh*
Lava is a set of Haskell modules that define a domain specific hardware description language for producing circuits for implementations on Field Programmable Gate Arrays (`http://www.xilinx.com`) (FPGAs). Previous work has focused on designing and implementing a robust and practical system for realizing structural (graph based) circuit descriptions using combinations (higher order functions or connection patterns) to compose circuits in interesting ways.
We have now turned our attention to capturing system level information in Lava. In particular we would like to describe the architecture of bus-based systems were components communicate not through directly connected wires but instead use a protocol for communication over a bus. Ideally we would like to define some kind of embedded type system that captures not only what information is communicated but how it is communicated (direct connection, bus, FIFO, shared memory, interrupt etc.). This could allow much higher level descriptions of "System on Chip" (SoC) systems and allows for the possibility of automatically generating interfacing circuitry. As our focus changes from computation (what the gates do) to communication (how to compose large modules) we expect to produce interesting requirements for a type system for hardware description languages intended for designing SoCs.

**Further reading:**

`http://www.xilinx.com/labs/lava/`

## 6.3 Haskell in Education

At the recent workshop on functional and declarative languages in education (FDPE 2002), there seemed to be a consensus that functional programming in first year computer science courses works best if it focusses on general programming and computer science concepts, i.e. uses functional programming as a tool, not as a goal or main topic.
While preparing the ground for a more ambitious initiative (creating a formally based software engineering program at his University), Rex Page has been collecting empirical evidence on using Haskell as a tool for familiarizing students with the idea of reasoning about software artifacts. Jose Labra introduces a new project to develop a generic web based programming environment to teach Haskell and other programming languages. See also the EDSL project at Yale (section 6.4.3), developing domain-specific languages for use in high-school education.

### 6.3.1 Beseme Project

**Report by:** *Rex Page*
Studying connections between programming effectiveness and practice in reasoning about software.
The test and debug cycle accounts for the entire defect prevention strategy in most software projects. What difference might it make if software developers had some experience in reasoning about software artifacts using methods from mathematical logic? The Beseme Project (three syllables, all rhyming with "eh") is gathering some evidence bearing on this question by introducing students in a discrete mathematics course to logic through examples based entirely on reasoning about software, most of which is expressed as Haskell equations. The subsequent performance of these students in a programming-intensive course is compared to that of students who have gone through a traditional discrete mathematics course. Progress reports and course materials including over 350 lecture slides, homework, exams, solutions, and software tools are available through the Beseme website:
`http://www.cs.ou.edu/~beseme/`

### 6.3.2 Idefix Project

**Report by:** *Jose Labra*
I have started a project called IDEFIX where we want to develop a generic web based programming environment to teach Haskell and other programming languages.
The first steps of the system were presented in the conference "Implementation of Functional Programming languages", 2002 (Madrid).

**Contact person:** Jose Emilio Labra Gayo `http://www.di.uniovi.es/~labra`

**Further reading:**

`http://www.di.uniovi.es/aplt/idefix`

## 6.4 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. Unfortunately, we don't seem to reach some of them yet, so if you're reading this, please make sure that your group is represented in the next edition!

### 6.4.1 Functional Programming at Chalmers

**Report by:** *Patrik Jansson*

Here is some input on Haskell-related research at CS.Chalmers.se. This is by no means complete - we do other Haskell-related stuff as well, including the work on generic programming (3.5) and QuickCheck (5.3.4) ...

A bigger project is just about to start: (in addition to the professors listed below, Marcin Benke, Koen Claessen, Patrik Jansson and Ulf Norell will also be involved)

**Combining Verification Methods in Software Development** *Thierry Coquand, Peter Dybjer, John Hughes, Mary Sheeran*

The goal of this programme is to develop methods for improving software quality. The approach is to integrate a variety of verification methods into a framework which permits a *smooth progression* from hacking code to fully formal proofs of correctness. By a pragmatic integration of different techniques in a next-generation program development tool, we hope to handle systems on a much larger scale than hitherto.

This proposal builds on and combines our extensive and internationally well-known research in interactive theorem provers, formal methods, program analysis and transformation, and automatic testing. Our long experience with functional languages, which we use both as implementation tools and a test-bed, improves our chances of success in tackling these difficult problems.

**Further reading:**

`http://www.cs.chalmers.se/Cs/Research/Functional/`

### 6.4.2 Formal Methods at Bremen University

**Report by:** *Christoph Lüth and Christian Maeder*

**Members:** Christoph Lüth, Klaus Lüttich, Christian Maeder, Achim Mahnke, Till Mossakowski, Markus Roggenbach, George Russell, Lutz Schröder

The activities of our group are centered on the UniForM workbench and the Common Algebraic Specification Language (CASL).

The **UniForM workbench** is an integration framework mainly geared towards tools for formal methods. It uses a simple, powerful and flexible notion of events to model all interactions between tools and users. In particular, the workbench provides HTk, an encapsulation of Tcl/Tk based on our event model (see HTk under Graphical User Interfaces). The workbench is actively used in the MMiSS project (`http://www.mmiss.de`). It currently contains about 70k lines of Haskell code (plus a few hundred lines of C), and compiles with the Glasgow Haskell Compiler, making use of many of its extensions, in particular concurrency.

We are also using GHC to develop tools, like parsers and static analysers, for languages from the **CASL** family.

Several parsers have been written using the combinator library Parsec. (Annotated) terms (from the ATerm Library) are used as a data exchange format and we use DrIFT (section 5.2.1) to derive instances for conversions. For various graph data structures we use the Functional Graph Library (FGL). Documentation will be generated using Haddock (section 5.3.5). (for Parsec, ATerm, and FGL, see `http://haskell.org/libraries/`)

One extension of CASL, namely **HasCASL**, strives to combine CASL and Haskell. The HasCASL development paradigm (from requirements to functional programs) has been presented at the recent IFL'02 Workshop in Madrid.

The language **HetCASL** is a combination of several specification languages used in formal methods, such as CSP, CASL, HasCASL, and Modal and Temporal Logic. We exploit Glasgow Haskell's multiparameter type classes and functional dependencies to provide a type-safe interface to analysis tools for particular languages. Specifications involving several languages can be processed using existential and dynamic types.

**Further reading:**

Group activities overview: `http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/`

UniForM workbench `http://www.informatik.uni-bremen.de/uniform/wb`

HTk `http://www.informatik.uni-bremen.de/htk`

CASL `http://www.informatik.uni-bremen.de/cofi`

### 6.4.3 The Yale Haskell Group

**Report by:** *John Peterson*

The members of our group are Paul Hudak, John Peterson, Henrik Nilsson, Antony Courtney, and Liwen Huang.

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). Particular examples are Frob (Functional Robotics) and FVision (Functional Vision). FRP was originally developed by Conal Elliott as part of the Fran animation system. It has three basic ideas: continuous- and discrete-time signals, functions from signals to signals, and switching. FRP is particularly useful in hybrid systems: applications that have both continuous time and discrete time aspects.

**Functional Reactive Programming**

**Yampa** is the culmination of our efforts to provide domain-specific embedded languages for the programming of hybrid systems. Yampa differs from previous FRP based system in that it is structured using the arrow combinators (section 3.7.2). This greatly reduces the chance of introducing space and time leaks into reactive, time-varying systems.

We have released a preliminary version of Yampa that contains:

- The Yampa Base Library, containing generic functions for the expression of continuous behaviors, discrete events, and the interactions between behaviors and events.
- The Yampa Robotics Library, containing entities tailored for controlling mobile robots, both real and simulated, in the style of Frob, our FRP-based robotics language. The simulator is written using Yampa's Base and HGL, the Haskell Graphics Library (section 4.4.1), and performs physical modelling of mobile differential-drive robots equipped with several kinds of sensors. A preconfigured version of the simulator allows one to play RoboCup Soccer.

With the Base Library and HGL (or any other graphics library), it is easy to write reactive animation programs in the style of Fran. Thus there is no need for a special library to support graphics and animation.

Antony Courtney is working on yet another graphics library for Haskell to provide capabilities similar to the Java 2-D graphics library.

**Domain Specific Languages for Education**

The goal of this project is to use computers to assist students in understanding fundamental concepts within the core of the high school (ages 12 - 18) curriculum. In our approach, students must be able to describe objects in a learning domain in a formal manner that looks suspiciously like Haskell code. These abstract objects may be mathematical functions, physical laws, computational processes, or other intangible entities. Once the computer knows what the student is expressing, it can then realize the object in ways that help the student to explore and understand its properties.

We are currently working with on languages: one for mathematical visualization (Pan) and another for algorithmic music composition (Haskore). We don't have an educational version of Haskore yet but have used it (and Hugs) on high school students with good results.

We are about to release a new version of Conal Elliott's Pan system, Pan#, in a version that no longer requires the user to use a Haskell compiler. We use a subset of Haskell ("friendly Haskell") supplemented with primitives for color manipulation to describe images. This system depends on C# and Microsoft's .NET at the moment. A "hackers" version of the software is available now on the web and a formal release should occur around December 1.

**Further reading:**

`http://www.haskell.org/yale`
`http://haskell.org/yampa`
`http://www.haskell.org/edsl`

### 6.4.4 Functional Programming at Brooklyn College, City University of New York

**Report by:** *Murray Gross*

One prong of the Metis Project at Brooklyn College, City University of New York, is research on and with Parallel Haskell (section 3.3.2) in a Mosix-cluster environment. In fact, although we are just starting up, we want to both use and work on Haskell–we are gathering a number of smaller research efforts under a single umbrella.

At the present time, with the assistance of the group at Heriot Watt University (Edinburgh), we are working on implementing the parallel run-time system (GUM) for release 5 of GHC. We would be most interested in forming additional cooperative relationships with other academic and industrial research partners.

**Contact:** Murray Gross, <magross@its.brooklyn.cuny.edu>

### 6.4.5 Functional Programming at Utrecht University

**Report by:** *Doaitse Swierstra*

**All UU Software** (`http://www.cs.uu.nl/groups/ST/`)
We are well on our way to make all our Haskell modules mutually consistent and to make them available through a CVS server at `cvs.cs.uu.nl`, in the directory `uust`. Currently included are our parser combinators, pretty printers and attribute grammar system. Further software will be added in the near future.

**Parser Combinators** *(Doaitse Swierstra, Arthur Baars, Rui Guerra)*
The current version of the parser combinators constructs an online result, in the sense that parts of the result can be accessed even when parsing has not yet finished. This is especially useful when parsing and processing large files of similar information. Furthermore error messages are displayed while parsing (using `unsafePerformIO`). The underlying mechanism for achieving this is relatively costly, although parsing speed is not much slower than that of parsers generated off line using Frown or Happy (section 5.2.2). We plan to construct a companion module (based on an earlier approach) that contains a more strict result, and which we expect to be running even faster. Furthermore the module structure may be changed by making it possible for the user of the library to tune the internals of the machine even more using classes. In order to make the everyday use of the combinators not suffer from these changes we have separated the interface and the extensions from the basic implementation, so future changes can relatively easily be made.

Furthermore three special modules were constructed, since they contain far more complex combinators, that may probably not be used by most people. Two of the combinators enable the construction of a parser that reorders the elements it has recognized (merging or permutation parsing) and keep

track of this reordering by returning a function that can be used to reconstruct the original order. Inspiration for this came from the wish to be able to record the original input in such a way that error messages can be easily added to it. The third module can be used to construct parsers for languages that follow the Haskell off side rule when parsing. This turned out to be quite complicated since the precise parsing rules have been defined in terms of parse errors, and our combinators have a standard way of handling such errors; as a consequence we had to afflict some brain-damage.

**Helium** *(Arjan van IJzendoorn, Bastiaan Heeren, Daan Leijen, Rijk-Jan van Haaften)*
The purpose of the Helium project is to construct a lightweight compiler for a subset of Haskell that is especially directed to beginning programmers. We try to give useful feedback for oft occurring mistakes. Included in the Helium compiler are the type error determining techniques described in the next section. One of the aspects of the compiler is that it also logs errors, so we can track the kind of problems students are having, and improve the error messages and hints. The compiler uses Daan Leijen's LVM (Lazy Virtual Machine) as back-end. The complete type checker and code generator has been constructed with our AG (attribute grammar system).
As a side effect of the Helium project a data type was defined for the internal representation of Haskell programs, around which we try to focus our activities.

**Improving Type Errors** *(Bastiaan heeren, Jurriaan Hage, Doaitse Swierstra)*
As everyone has experienced it is not always the case that error messages coming from the type inferencer point at the point in the program where the actual corrections have to be made. We constructed a constraint based solver that is used to pinpoint the most likely location. We will extend this solver with the possibility to oft occurring erroneous patterns, that stem from commonly made mistakes, in order to provide more specific feedback.

**The attribute grammar system AG** *(Arthur Baars, Doaitse Swierstra)*
The system has been bootstrapped, and now provides extensive error messages in case the attribute grammar contains errors. Only the type checking of the semantic functions is postponed to the Haskell compiler that is processing the output of the system. In a newer version we have added the conventional data flow analyses, so we may point at circularities, and can do experiments with generating more strict evaluators, of which we hope they will run even faster. The system is used in the course on Implementation of Programming Languages.

**Type Checker for Extended Haskell** *(Atze Dijkstra, Doaitse Swierstra)*
As a companion to Mark Jones' "Typing Haskell in Haskell" we are constructing a type inferencer for full (extended) Haskell. Some of its features are a consistent way of handling existential and polymorphic types, and the use of polymorphic kinds (if you want to know what they are good for read the "Typing Dynamic Typing" paper presented at the ICFP2002, `http://www.cs.uu.nl/people/arthurb/dynamic.html`). We are currently rounding of this construction with the less interesting, but more laborious parts of full Haskell. We plan to use this material later this year in a course on "Type Systems".

**Pretty Printing** *(Pablo Azero, Doaitse Swierstra)*
Our pretty printing combinators have been silently doing their work over the years. Currently we are updating them, so they can be generated by the new version of the AG system. They too will have a more flexible interface allowing naming of subformats by using a monadic top layer.

**Proxima** *(Martijn Schrage, Johan Jeuring, Lambert Meertens, Doaitse Swierstra)*
Proxima is a generic graphical structure editor with support for free editing (ie. normal typing instead of selecting transformations from menus) and computations over the data structure. The system has a layered architecture, which is described and implemented using a library of architecture combinators. For the presentation of the document data structure, the graphical presentation combinator library Xprez has been developed. The user interface is implemented using the ported ObjectIO library (section 4.3.2), stemming from Clean.
One of the intended applications of Proxima is an editor/IDE for the language Helium. It will support editable pretty printed code in which types and error messages can be shown. Sources can be edited by normal typing (also for changing layout) as well as by performing structural edit commands.
A prototype of Proxima is expected to be ready in the first half of 2003.

**Syntax Macros** *(Arthur Baars, Doaitse Swierstra)*
The syntax macros are now in a state that one gets a macro mechanism for free when using our attribute grammar system and parser combinators in constructing a front end of a compiler. Most of the necessary glueing code is automatically generated. The syntax macros make it possible to extend the context free grammar of a language on a per program basis. Examples of constructs that no longer have to be part of the standard language, but could have been defined using our macro mechanism are the `do`-notation, and the notation for list comprehensions. An open question, on which we work, is how to provide feedback to the user in terms of his original program. The current version is available at: `http://www.cs.uu.nl/people/arthurb/macros.html`

**C-front end and Type Checker** *(Alexey Rodriguez)*
As part of an abstract interpreter for C a parser and type checker for C have been produced. They are being brought in line with the modules in the CVS repository, and will appear there when consistent. The parser is interesting because it

shows how to deal with some weird aspects of the C context free grammar (parsing e.g. `a * b`).

**First Class Attribute Grammars** *(Arthur Baars, Doaitse Swierstra)*
We are investigating how to make language definitions more compositional, and how to capture recurring patterns of analysis and dat flow in compilers. Ideally we should like to have so-called first class aspects. It is a matter of research however how to integrate type checking and aspect oriented programming. Attempts using extendible records almost seem to do the job, but unfortuantely incorrect use leads to pages of error messages. We hope that following the techniques explained in `http://www.cs.uu.nl/people/arthurb/dynamic.html` may help to solve the problem.

### 6.4.6 Functional Programming at UKC

**Report by:** *Claus Reinke*
Here at the University of Kent at Canterbury, the functional programming interest group now includes about a dozen people (yes, we're growing;-) pursuing research interests in functional programming. Haskell is a major focus of teaching and research, although we also look at other languages (such as Erlang `http://www.erlang.org`;-).
Most of our Haskell projects have now moved into their own sections, such as the Haskell binding for Gtk2 (Axel Simon; 4.3.4), the Metrics and Visualization project Medina (Chris Ryder; 4.5.1), the functional refactoring project (Simon Thompson, Claus Reinke, Huiqing Li; 5.3.3), or the project combining functional programming and virtual worlds (Claus Reinke; 4.4.3).
**Keith Hanna** is continuing work on Vital, an implementation of Haskell intended for end-users in other disciplines (engineering, finance, maths, etc.). Its distinguishing features are that Haskell values are represented graphically in a workspace, that their look-and-feel is determined by a stylesheet, and that these values may be edited graphically (with changes being reflected back in the original Haskell program). The system was demonstrated at the recent ICFP conference in Pittsburgh (there's a paper in the proceedings as well).
At present, work is concentrated on allowing parametrised stylesheets so that, for instance, the look-and-feel of the elements of a datastructure can be varied independently of that of the overall datastructure. The next release of Vital is planned for mid 2003.

**Further reading:**

FP group:
`http://www.cs.ukc.ac.uk/research/groups/tcs/fp/`
Vital:
`http://www.cs.ukc.ac.uk/people/staff/fkh/Vital/`
Gtk2HS: `http://gtk2hs.sourceforge.net/`
FunWorlds:
`http://www.cs.ukc.ac.uk/people/staff/cr3/FunWorlds/`
Haskell metrics:
`http://www.cs.ukc.ac.uk/people/rpg/cr24/medina/`
Refactoring Functional Programs:
`http://www.cs.ukc.ac.uk/research/groups/tcs/fp/Refactor/`

### 6.4.7 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
(Royal Bank of Scotland) Dominic Steinitz

**Contact:** Dan Russell <D.Russell@kingston.ac.uk>
This community 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 has been further developed to include asynchronous processes (using Concurrent Haskell).
Libraries for the LDAP, ASN.1 and BER will be made available as open source by the end of November 2002. Future plans include analysis of other internet applications from a concurrency and mobility perspective to inform functional library designs.
Work has also started on a Case Tool to support a functional methodology.

**Further reading:**

FP Group: `http://www.kingston.ac.uk/~ku07009/Research/fpres.html`
Chris Reade: `http://www.kingston.ac.uk/~ku07009/`

## 6.5 Individual Haskellers

"What are you using Haskell for?" – the implementation mailing lists are full of people sending in bug reports and feature suggestions, stretching the implementations to their limits. Judging from the "reduced" examples sent in to demonstrate problems, there must be quite a few Haskell applications out there that haven't been announced anywhere (probably because Haskell is "just" the tool, not the focus of those projects).
If you're one of those serious Haskell users, why not write a sentence or two about your application? We'd be particularly interested in your experience with the existing tools (e.g., that

all-time-favourite: how difficult was it to tune the resource usage to your needs, after you got your application working? Which tools/libraries where useful to you? What is missing?).

---

**Hal Daume** <hdaume@ISI.EDU> writes: I use Haskell for statistical natural language processing: specifically, automatic document summarization (machine learning for how to do summarization by reading pairs of documents and human-created summaries of them). This will undoubtedly be the topic of my thesis. I have also implemented an unsupervised learning system in Haskell, available off my web page.

I also use Haskell for Haskell's sake and am currently working on a project which will convert Haskell code into (strict) ML code. It is not meant to be "industrial strength" but already it will convert the larger part of the Prelude. It is intended to allow analysis of under what circumstances laziness hurts us.

---

**Tom Pledger** <Tom.Pledger@peace.com> writes:

"Since 2001 I've mainly been working on a Haskell-like language for business data processing. It has a lot in common with discrete Functional Reactive Programming. A prototype of the runtime virtual machine, implemented in Haskell, passed some initial tests in April 2002. Now I'm concentrating on the database interface and the front end of the interpreter, and sketching some ideas for an Integrated Development Environment.

I take care not to ask for related free advice on the Haskell mailing lists, because my employer expects to own and profit from what I produce."

---

In our previous edition, **Johannes Waldmann** <joe@isun.informatik.uni-leipzig.de> reported on two projects in the context of his university teaching (not only Haskell), autotool (automatic assessment of theoretical computer science homework via Haskell DSELs) `http://theopc.informatik.uni-leipzig.de/~autotool/`, and a framework for board game programming contests (latest instance: `http://theopc.informatik.uni-leipzig.de/~joe/phutball/`). Both are still in use, but haven't seen any new development over the last six months. The proceedings of the recent FDPE 2002 workshop (1.4) include a paper on autotool.

---

**Oleg Kiselyov** <oleg@pobox.com> continues to extend his collection of Haskell programming miscellanea (we mentioned it in the tips&tricks section last time), exploring algorithms and programming techniques, with extensively commented example code `http://pobox.com/~oleg/ftp/Haskell/misc.html`. One recent addition is a discussion of programming with cyclic data structures "Pure-functional transformations of cyclic graphs and the Credit Card Transform." – illustrated by considering the problem of printing out a non-deterministic finite automaton (NFA) and transforming it into a deterministic finite automaton (DFA). Both NFA and DFA are represented as cyclic graphs.

"I'd also like to point out a Haskell project that I'll be developing further and further: a pure-functional lambda-calculator:
`http://pobox.com/~oleg/ftp/Haskell/Lambda_calc.lhs`
The present calculator implements what seems to be an efficient and elegant algorithm of normal order reductions. The algorithm is "more functional" than the traditionally used approach. The algorithm seems identical to that employed by yacc sans one critical difference. The calculator also takes a more "functional" approach to the hygiene of beta-substitutions, which is achieved by coloring of identifiers where absolutely necessary. This approach is "more functional" because it avoids a "global" counter or the threading of the paint bucket through the whole the process. The integration of the calculator with Haskell lets us store terms in variables and easily and intuitively combine them.

The calculator is fully functional (pun intended). I use it routinely to play with Lambda Calculus. One interesting twist is that lambda-calculus reductions seem to be closely related to LALR-like parsing."

---

**Mike Thomas** <mthomas@gil.com.au> writes: "I'm working on small bindings to the MPICH (open source parallel processing by message passing) and Proj (open source map projection) libraries in conjunction with a Haskell library to read, write, display and process GRASS (an open source geographic information system) mapsets and of doing some geochemical modelling, hopefully with the ability to distribute large map computations with MPICH.

The Windows version of GHC is my compiler of choice and I rely heavily on the Parsec and ObjectIO libraries, each of which I believe to be excellent development tools.

In the last HC&A Report, work had just commenced and continues in whatever non-family, non-work, non-other-project time I can find. My focus has been on low-level libraries, refining the handling of large images and adding support for more GRASS data formats. The next step (amongst many) is to make a map description data format and to use it to make nice maps from GRASS data. This work is not available on the web as it is relatively incomplete and I have not decided on a release model. More information on GRASS may be found at:
`http://grass.itc.it`
And MPICH:
`http://www-unix.mcs.anl.gov/mpi/mpich/`