Haskell Communities and Activities Report

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

Nineteenth Edition — November 2010

Janis Voigtländer (ed.)
Andreas Abel
Robin Adams
Iain Alexander
Krasimir Angelov
Heinrich Apfelmus
Jim Apple
Dmitry Astapov
Christiaan Baaij
Justin Bailey
Alexander Bau
Doug Beardsley
Jean-Philippe Bernardy
Tobias Bexelius
Annette Bieniusa
Mario Blažević
Anthonin Bonnefoy
Edwin Brady
Gwern Branwen
Joachim Breitner
Erik de Castro Lopo
Roman Cheplyaka
Olaf Chitil
Duncan Coutts
Simon Cranshaw
Nils Anders Danielsson
Dominique Devriese
Daniel Díaz
Larry Diehl
Atze Dijkstra
Jonas Duregård
Marc Fontaine
Patai Gergely
Brett G. Giles
Andy Gill
George Giorgidze
Dmitry Golubovsky
Carlos Gomez
Matthew Gruen
Torsten Grust
Jurriaan Hage
Sönke Hahn
Bastiaan Heeren
Judah Jacobson
Jeroen Janssen
David Himmelstrup
Guillaume Hoffmann
Martin Hofmann
Jasper Van der Jeugt
Farid Karimipour
Oleg Kiselyov
Lennart Kolmodin
Michal Konečný
Eric Kow
Ben Lippmeier
Andres Löh
Tom Lokhorst
Rita Loogen
Ian Lynagh
John MacFarlane
Christian Maeder
José Pedro Magalhães
Ketil Malde
Vivian McPhail
Arie Middelkoop
Ivan Lazar Miljenovic
Neil Mitchell
Dino Morelli
JP Moresmau
Matthew Naylor
Victor Nazarov
Jürgen Nicklisch-Franken
Rishiyur Nikhil
Thomas van Noort
Johan Nordlander
Miguel Pagano
Jens Petersen
Simon Peyton Jones
Bernie Pope
Matthias Reisner
Alberto Ruiz
David Sabel
Antti Salonen
Ingo Sander
Uwe Schmidt
Martijn Schrage
Tom Schrijvers
Jeremy Shaw
Marco Silva
Axel Simon
Michael Snoyman
Will Sonnex
Martijn van Steenbergen
Martin Sulzmann
Doaitse Swierstra
Henning Thielemann
Simon Thompson
Thomas Tuegel
Marcos Viera
Janis Voigtländer
Jan Vornberger
David Waern
Gregory D. Weber
Stefan Wehr
Mark Wotton
Kazu Yamamoto
Brent Yorgey

Preface

This is the 19th edition of the Haskell Communities and Activities Report. As usual, fresh entries are formatted using a blue background, while updated entries have a header with a blue background. Entries for which I received a liveness ping, but which have seen no essential update for a while, have been replaced with online pointers to previous versions. Other entries on which no new activity has been reported for a year or longer have been dropped completely. Please do revive such entries next time if you do have news on them.

I have restructured the report a bit. Any feedback on that, or on my further attempts to improve the generation of the html version of the report (see http://haskell.org/communities/11-2010/html/report.html), or in fact on anything else, would be very welcome.

A call for new entries and updates to existing ones will be issued on the usual mailing lists in April. Now enjoy the current report and see what other Haskellers have been up to lately.

Janis Voigtländer, University of Bonn, Germany, ⟨hcar@haskell.org⟩

Contents

1 Community

1.1 Haskellers

Report by:	Michael Snoyman
Status:	experimental

In the beginning of October, Haskellers was launched. It is a site designed to promote Haskell as a language for use in the real world by being a central meeting place for the myriad talented Haskell developers out there. It allows users to create profiles complete with skill sets and packages authored and gives employers a central place to find Haskell professionals.

Though the site is still in its infancy, the response has been staggering. Within a week of its launch, we are now sitting at over 200 active accounts. We are still planning on lots of new features: we may be adding social networking functionality, job postings, user polls, and much more. If you have any ideas, please let me know. And if you are at all involved in the Haskell community, be sure to create a profile.

Further reading

http://www.haskellers.com/

1.2 Haskell Wikibook

Report by:	Heinrich Apfelmus
Participants:	Duplode, Orzetto, David House, Eric Kow, and other contributors
Status:	active development

The goal of the Haskell Wikibook project is to build a community textbook about Haskell that is at once free (as in freedom and in beer), gentle, and comprehensive. We think that the many marvelous ideas of lazy functional programming can and thus should be accessible to everyone in a central place. In particular, the Wikibook aims to answer all those conceptual questions that are frequently asked on the Haskell mailing lists.

Everyone including you, dear reader, are invited to contribute, be it by spotting mistakes and asking for clarifications or by ruthlessly rewriting existing material and penning new chapters.

Thanks to user Duplode, a major reorganization of the introductory chapters is in progress.

Further reading

http://en.wikibooks.org/wiki/Haskell

1.3 Cartesian Closed Comic

Report by:	Roman Cheplyaka
Participants:	Maria Kovalyova

See: http://haskell.org/communities/05-2010/html/report.html#sect1.6.

2 Articles/Tutorials

2.1 The Monad.Reader

Report by: Brent Yorgey

There are plenty of academic papers about Haskell and plenty of informative pages on the HaskellWiki. Unfortunately, there is not much between the two extremes. That is where The Monad.Reader tries to fit in: more formal than a Wiki page, but more casual than a journal article.

There are plenty of interesting ideas that maybe do not warrant an academic publication—but that does not mean these ideas are not worth writing about! Communicating ideas to a wide audience is much more important than concealing them in some esoteric journal. Even if it has all been done before in the Journal of Impossibly Complicated Theoretical Stuff, explaining a neat idea about "warm fuzzy things" to the rest of us can still be plain fun.

The Monad.Reader is also a great place to write about a tool or application that deserves more attention. Most programmers do not enjoy writing manuals; writing a tutorial for The Monad.Reader, however, is an excellent way to put your code in the limelight and reach hundreds of potential users.

Since the last HCAR there has been one new issue, featuring an article on combinators for automata, an interatee tutorial, and an exploration of priority queue implementations. The next issue will be published in November.

Further reading

http://themonadreader.wordpress.com/

2.2 Oleg's Mini Tutorials and Assorted Small Projects

Report by: Oleg Kiselyov

The collection of various Haskell mini tutorials and assorted small projects (http://okmij.org/ftp/Haskell/) has received two additions:

Eliminating Existentials

The web page demonstrates various ways of eliminating explicit existential quantification in data types, replacing such data types with isomorphic simple, first-order types. Although such a replacement is of most use in the languages like SML without direct support for existentials, the technique may simplify Haskell programs as well, reducing their reliance on non-standard extensions. The web page uses Haskell extensively to explain the technique and to demonstrate its correctness, by writing isomorphisms between existentials and simple-type representations.

The most interesting case was the elimination of a translucent existential, which exposed part of its structure, being a list. The type of the elements of the list is opaque. Eliminating such existential required nested data types.

The web page discusses several ways of collecting values of different types in the same list, stressing open unions implemented with and without existentials. Implicitly heterogeneous lists without existentials offer no value abstraction, let alone type abstraction. On the upside, these open unions support a projection operation, or safe downcast.

http://okmij.org/ftp/Computation/Existentials.html

Type-class overloaded functions: second-order typeclass programming with backtracking

We describe functions polymorphic over classes of types. Each instance of such (2-polymorphic) function uses ordinary 1-polymorphic methods, to generically process values of many types, the members of that 2-instance type class. The typeclass constraints are thus manipulated as first-class entities. We also show how to write typeclass instances with back-tracking: if one instance does not apply, the typechecker will chose the 'next' instance — in the precise meaning of 'next'.

We show a method to describe classes of types in a concise way: instead of the exhaustive enumeration of class members, we use unions, class differences, and unrestricted comprehension. These classes of types may be either closed or open (extensible). After the classes are defined, we can write arbitrarily many functions overloaded over these type classes. An instance of our function for a specific type class may use polymorphic functions to generically process all members of that type class. Our functions are hence second-order polymorphic.

http://okmij.org/ftp/Haskell/types.html#poly2

2.3 Haskell Cheat Sheet

Report by: Justin Bailey
Status: active development

The "Haskell Cheat Sheet" covers the syntax, keywords, and other language elements of Haskell 98. Beginning to intermediate Haskell programmers should find it useful; it can even serve as a memory aid for experts.

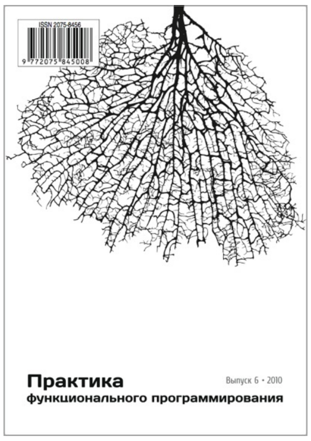
The cheat sheet can be downloaded directly from http://cheatsheet.codeslower.com or installed using cabal (`cabal install cheatsheet`). Spanish and Japanese translations of the cheatsheet can be found on the web site, as well.

Further reading

http://cheatsheet.codeslower.com

2.4 Practice of Functional Programming

Report by:	Dmitry Astapov
Status:	six issues ready, issue #7 is in pipeline, collecting materials for more



"Practice of Functional Programing" is a Russian electronic magazine promoting functional programming. The magazine features articles that cover both theoretical and practical aspects of the craft. Most of the already published material is directly related to Haskell.

The magazine attempts to keep a bi-monthly release schedule, with Issue #7 slated for release early in 2011.

Full contents of current and past issues are available in PDF from the official site of the magazine free of charge.

Articles are in Russian, with English annotations.

Further reading

http://fprog.ru/ for issues ##1–6

3 Implementations

3.1 Haskell Platform

Report by: Duncan Coutts

Background

The Haskell Platform (HP) is the name of the “blessed” set of libraries and tools on which to build further Haskell libraries and applications. It takes a core selection of packages from the more than 2500 on Hackage (→ 6.7.1). It is intended to provide a comprehensive, stable, and quality tested base for Haskell projects to work from.

Historically, GHC shipped with a collection of packages under the name `extralibs`. Since GHC 6.12 the task of shipping an entire platform has been transferred to the Haskell Platform.

Recent progress

During the summer we had the second major release of the platform. This is the 2010.2.0.x release series. While there were no new packages included in this major release, there have been a few significant upgrades including QuickCheck version 2, the latest versions of the ‘regex-*’ packages and of course GHC 6.12.x.

Looking forward

Major releases take place on a 6 month cycle. The next major release will be in January 2011 and — barring any major problems — will include GHC 7.0.x.

This is the first round where we have started to use the new procedure for adding packages. There were two proposals: one to add the ‘text’ package and another for a major update to the ‘mtl’ library. At the time of writing the final decision has not been made on whether these proposals will be accepted for this round.

For the following major release, we would like to invite package authors to propose new packages. We also invite the rest of the community to take part in the review process on the libraries mailing list `libraries@haskell.org`. The procedure involves writing a package proposal and discussing it on the mailing list with the aim of reaching a consensus. Details of the procedure are on the development wiki.

Further reading

http://haskell.org/haskellwiki/Haskell_Platform
- Download: http://hackage.haskell.org/platform/
- Wiki: http://trac.haskell.org/haskell-platform/
- Adding packages: http://trac.haskell.org/haskell-platform/wiki/AddingPackages

3.2 The Glasgow Haskell Compiler

Report by: Simon Peyton Jones
Participants: many others

GHC is humming along. We are currently deep into the release cycle for GHC 7.0. We have finally bumped the major version number, because GHC 7.0 has quite a bit of new stuff:

- As long promised, Simon PJ and Dimitrios have spent a good chunk of the summer doing a **complete rewrite of the constraint solver in the type inference engine**. Because of GHC’s myriad type-system extensions, especially GADTs and type families, the old engine had begun to resemble the final stages of a game of Jenga. It was a delicately-balanced pile of blocks that lived in constant danger of complete collapse, and had become extremely different to modify (or even to understand). The new inference engine is much more modular and robust; it is described in detail in our paper [OutsideIn]. A blog post describes some consequential changes to let generalisation [LetGen].

  As a result we have closed dozens of open type inference bugs, especially related to GADTs and type families.

- There is a new, **robust implementation of INLINE pragmas** that behaves much more intuitively. GHC now captures the original RHS of an INLINE function, and keeps it more-or-less pristine, ready to inline at call sites. Separately, the original RHS is optimised in the usual way. Suppose you say

  ```
  {-# INLINE f #-}
  f x = ...blah...

  g1 y = f y + 1
  g2 ys = map f ys
  ```

  Here, `f` will be inlined into `g1` as you would expect, but obviously not into `g2` (since it is not applied to anything). However `f`’s right hand side will be optimised (separately from the copy retained for inlining) so that the call from `g2` runs optimised code.

  There is a raft of other small changes to the optimisation pipeline too. The net effect can be dramatic: Bryan O’Sullivan reports some five-fold (!) improvements in his text-equality functions, and concludes

"The difference between 6.12 and 7 is so dramatic, there's a strong temptation for me to say 'wait for 7!' to people who report weaker than desired performance."

◦ David Terei implemented a new **back end for GHC using LLVM**. In certain situations using the LLVM backend can give fairly substantial performance improvements to your code, particularly if you are using the Vector libraries, DPH or making heavy use of fusion. In the general case it should give as good performance or slightly better than GHC's native code generator and C backend. You can use it through the `-fllvm` compiler flag. More details of the backend can be found in David's and Manuel Chakravarty's Haskell Symposium paper [Llvm].

◦ Bryan O'Sullivan and Johan Tibell implemented a new, **highly-concurrent I/O manager**. GHC now supports over a hundred thousand open I/O connections. The new I/O manager defines a separate backend per operating system, using the most efficient system calls for that particular operating system (e.g., `epoll` on Linux). This means that GHC can now be used to implement servers that make use of, e.g., HTTP long polling, where the server needs to handle a large number of open idle connections.

◦ In joint work with Phil Trinder and his colleagues at Herriot Watt, Simon M designed and implemented a new **parallel strategies library**, described in their 2010 Haskell Symposium paper [Seq].

◦ As reported in the previous status update, **the runtime system has undergone substantial changes** to the implementation of lazy evaluation in parallel, particularly in the way that threads block and wake up again. Certain benchmarks show significant improvements, and some cases of wildly unpredictable behaviour when using large numbers of threads are now much more consistent.

◦ The **API for asynchronous exceptions** has had a redesign. Previously the combinators `block` and `unblock` were used to prevent asynchronous exceptions from striking during critical sections, but these had some serious disadvantages, particularly a lack of modularity where a library function could unblock asynchronous exceptions despite a prevailing `block`. The new API closes this loophole, and also changes the terminology: preventing asynchronous exceptions is now called "masking", and the new combinator is `mask`. See the documentation for the new API in `Control.Exception` for more details.

We are fortunate to have a growing team of people willing to roll up their sleeves and help us with GHC. Amongst those who have got involved recently are:

◦ Daniel Fischer, who worked on improving the performance of the numeric libraries.
◦ Milan Straka, for great work improving the performance of the widely-used containers package [Containers].
◦ Greg Wright is leading a strike team to make GHC work better on Macs, and has fixed the RTS linker so that GHCi will now work in 64-bit mode on OS X.
◦ Evan Laforge, who has taken on some of the long-standing issues with the Mac installer.
◦ Sam Anklesaria implemented full import syntax for GHCi, and rebindable syntax for conditionals.
◦ PHO, who improved the OS X support.
◦ Sergei Trofimovich, who has fixed GHC on some less common Linux platforms.
◦ Marco Túlio Gontijo e Silva, who has been working on the RTS.
◦ Matthias Kilian, who has been working on *BSD support.
◦ Dave Peixotto, who has improved the PAPI support.
◦ Edward Z. Yang, who has implemented interruptible FFI calls.
◦ Reiner Pope, who added view patterns to Template Haskell.
◦ Gabor Pali, who added thread affinity support for FreeBSD.
◦ Bas van Dijk has been improving the exceptions API.

At GHC HQ we are having way too much fun; if you wait for us to do something you have to wait a long time. So do not wait; join in!

Language developments, especially types

GHC continues to act as an incubator for interesting new language developments. Here is a selection that we know about:

◦ José Pedro Magalhães is implementing the **derivable type classes** mechanism (→ 6.4.1) described in his 2010 Haskell Symposium paper [Derivable]. I plan for this to replace GHC's current derivable-type-class mechanism, which has a poor power-to-weight ratio and is little used.

◦ Stephanie Weirich and Steve Zdancewic had a great sabbatical year at Cambridge. One of the things we worked on, with Brent Yorgey who came as an intern, was to close the embarrassing hole in the type system concerning **newtype deriving** (see Trac bug #1496). I have delayed fixing until I could figure out a Decent Solution, but now we know; see our 2011 POPL paper [Newtype]. Brent is working on some infrastructal changes to GHC's Core language, and then we will be ready to tackle the main issue.

◦ Next after that is a mechanism for **promoting types to become kinds**, and data constructors to become types, so that you can do *typed* functional programming at the type level. Conor McBride's SHE prototype is the inspiration here [SHE]. Currently it is, embarrassingly, essentially untyped.

◦ **Template Haskell** seems to be increasingly widely used. Simon PJ has written a proposal for a raft of improvements, which we plan to implement in the new year [TemplateHaskell].

◦ Iavor Diatchki plans to add **numeric types**, so that you can have a type like `Bus 8`, and do simple arithmetic at the type level. You can encode this stuff, but it is easier to use and more powerful to do it directly.

◦ David Mazieres at Stanford wants to implement **Safe Haskell**, a flag for GHC that will guarantee that your program does not use `unsafePerformIO`, foreign calls, RULES, and other stuff.

7.0 also has support for the Haskell 2010 standard, and the libraries that it specifies.

Packages and the runtime system

◦ Simon Marlow is working on a new garbage collector that is designed to improve scaling of parallel programs beyond small numbers of cores, by allowing each processor core to collect its own local heap independently of the other cores. Some encouraging preliminary results were reported in a blog post. Work on this continues; the complexity of the system and the number of interacting design choices means that achieving an implementation that works well in a broad variety of situations is proving to be quite a challenge.

◦ The "new back end" is still under construction. This is a rewrite of the part of GHC that turns STG syntax into C–, i.e., the bit between the Core optimisation passes and the native code generator. The rewrite is based on [Hoopl], a data-flow optimisation framework. Ultimately this rewrite should enable better code generation. The new code generator is already in GHC, but turned off by default; you get it with the flag `-fuse-new-codegen`. Do not expect to get better code with this flag yet!

The Parallel Haskell Project

Microsoft Research is funding a 2-year project to develop the real-world use of parallel Haskell. The project has recently kicked off with four industrial partners, with consulting and engineering support from Well-Typed (→ 10.1). Each organisation is working on its own particular project making use of parallel Haskell. The overall goal is to demonstrate successful serious use of parallel Haskell, and along the way to apply engineering effort to any problems with the tools that the organisations might run into.

We will shortly be announcing more details about the partner organisations and their projects. For the most part the projects are scientific and focus on single-node SMP systems, though one of the partners is working on network servers and another partner is very interested in clusters. In collaboration with Bernie Pope, the first tangible results from the project will be a new MPI binding (→ 5.1.2), which will appear on hackage shortly.

Progress on the project will be reported to the community. Since there are now multiple groups in the community that are working on parallelism, the plan is to establish a parallel Haskell website and mailing list to provide visibility into the various efforts and to encourage collaboration.

Data Parallel Haskell

Since the last report, we have continued to improve support for nested parallel divide-and-conquer algorithms. We started with QuickHull and are now working on an implementation of the Barnes-Hut *n*-body algorithm. The latter is not only significantly more complex, but also requires the vectorisation of recursive tree data-structures, going well beyond the capabilities of conventional parallel-array languages. In time for the stable branch of GHC 7.0, we replaced the old, per-core sequential array infrastructure (which was part of the sub-package `dph-prim-seq`) by the vector package — vector started its life as a next-generation spin off of `dph-prim-seq`, but now enjoys significant popularity independent of DPH.

The new handling of INLINE pragmas as well as other changes to the Simplifier improved the stability of DPH optimisations (and in particular, array stream fusion) substantially. However, the current candidate for GHC 7.0.1 still contains some performance regressions that affect the DPH and Repa libraries and to avoid holding up the 7.0.1 release, we decided to push fixing these regressions to GHC 7.0.2. More precisely, we are planning a release of DPH and Repa that is suitable for use with GHC 7.0 for the end of the year, to coincide with the release of GHC 7.0.2. From GHC 7.0 onwards, the library component of DPH will be shipped separately from GHC itself and will be available to download and install from Hackage as for other libraries.

To catch DPH performance regressions more quickly in the future, Ben Lippmeier implemented a performance regression testsuite that we run nightly on the HEAD. The results can be enjoyed on the GHC developer mailing list.

Sadly, Roman Leshchinskiy has given up his full-time engagement with DPH to advance the use of Haskell in the financial industry. We are looking forward to collaborating remotely with him.

Installers

The GHC installers have also received some attention for this release.

The Windows installer includes a much more up-to-date copy of the MinGW system, which in particular fixes a couple of issues on Windows 7. Thanks to Claus Reinke, the installer also allows more control over the registry associations etc.

Meanwhile, the Mac OS X installer has received some attention from Evan Laforge. Most notably, it is now possible to install different versions of GHC side-by-side.

Bibliography

Containers "The performance of the Haskell containers package", Straka, Haskell Symposium 2010.

Derivable "A generic deriving mechanism for Haskell", Magalhães, Dijkstra, Jeuring and Löh, Haskell Symposium 2010.

LetGen "Let generalisation in GHC 7.0", Peyton Jones, blog post Sept 2010.

Newtype "Generative Type Abstraction and Type-level Computation", Weirich, Zdancewic, Vytiniotis, and Peyton Jones, POPL 2011.

Llvm "An LLVM Backend for GHC", Terei and Chakravarty, Haskell Symposium 2010.

OutsideIn "Modular type inference with local assumptions: OutsideIn(X) ", Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann, Draft.

Seq "Seq no more", Marlow, Maier, Trinder, Loidl, and Aswad, Haskell Symposium 2010.

SHE The Strathclyde Haskell Enhancement, Conor McBride, 2010.

TemplateHaskell New directions for Template Haskell, Peyton Jones, blog post October 2010.

Hoopl Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation.

3.3 LHC

Report by:	David Himmelstrup
Participants:	Austin Seipp
Status:	active development

LHC is a backend for the Glorious Glasgow Haskell Compiler (→3.2), adding low-level, whole-program optimization to the system. It is based on Urban Boquist's GRIN language, and using GHC as a frontend, we get most of its great extensions and features.

Essentially, LHC uses the GHC API to convert programs to external core format — it then parses the external core, and links all the necessary modules together into a whole program for optimization. We currently have our own base library (heavily and graciously taken from GHC). This base library is similar to GHC's (module-names and all), and it is compiled by LHC into external core and the package is stored for when it is needed. This also means that if you can output GHC's external core format, then you can use LHC as a backend.

The short-term goal is to make LHC faster, easier to use, and more complete in its coverage of Haskell 98.

Further reading

- http://lhc.seize.it/
- http://lhc-compiler.blogspot.com/

3.4 The Helium Compiler

Report by:	Jurriaan Hage
Participants:	Bastiaan Heeren, Arie Middelkoop

See: http://haskell.org/communities/05-2009/html/report.html#sect2.3.

3.5 UHC, Utrecht Haskell Compiler

Report by:	Atze Dijkstra
Participants:	many others
Status:	active development

What is new? UHC is the Utrecht Haskell Compiler, supporting almost all Haskell98 features and most of Haskell2010, plus experimental extensions. Recently version 1.1.0 was released, featuring generic deriving (→6.4.1), a new configurable garbage collector, initial support for building with UHC via cabal, and many bug fixes.

We plan the next release to offer a Javascript backend. Furthermore we hope to add optimizations to eliminate some of the obvious inefficiencies. As part of this work the intent is also to better integrate the work done on whole program analysis.

UHC Blog Recently a UHC blog has been started. The intent is to scribble about the internals of UHC and issues arising out of the implementation.

What do we currently do and/or has recently been completed? As part of the UHC project, the following (student) projects and other activities are underway (in arbitrary order):

- Jeroen Bransen (PhD): “Incremental Global Analysis” (starting up).
- Jan Rochel (PhD): “Realising Optimal Sharing”, based on work by Vincent van Oostrum and Clemens Grabmayer.

- Arie Middelkoop (PhD): type system formalization and automatic generation from type rules.
- Tom Lokhorst: type based static analyses (recently completed, available with proper configuration).
- Jeroen Leeuwestein: incrementalization of whole program analysis.
- Atze van der Ploeg: lazy closures (recently completed).
- Paul van der Ende: garbage collection & LLVM (recently completed, available with proper configuration).
- Jeroen Fokker: GRIN backend, whole program analysis.
- Călin Juravle: base libraries (completed upto Haskell98, integrated).
- Levin Fritz: base libraries for Java backend (completed, integrated).
- Andres Löh: Cabal support (completed initial support, integrated).
- José Pedro Magalhães: generic deriving ((→ 6.4.1), completed, integrated, presented at Haskell Symposium).
- Doaitse Swierstra: parser combinator library.
- Atze Dijkstra: overall architecture, type system, bytecode interpreter + java + javascript backend, garbage collector.

Background UHC actually is a series of compilers of which the last is UHC, plus infrastructure for facilitating experimentation and extension. The distinguishing features for dealing with the complexity of the compiler and for experimentation are (1) its stepwise organisation as a series of increasingly more complex standalone compilers, the use of DSL and tools for its (2) aspectwise organisation (called Shuffle) and (3) tree-oriented programming (Attribute Grammars, by way of the Utrecht University Attribute Grammar (UUAG) system (→ 5.4.1).

Further reading

- UHC Homepage: http://www.cs.uu.nl/wiki/UHC/WebHome
- UHC Blog: http://utrechthaskellcompiler.wordpress.com
- Attribute grammar system: http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem
- Parser combinators: http://www.cs.uu.nl/wiki/HUT/ParserCombinators
- Shuffle: http://www.cs.uu.nl/wiki/Ehc/Shuffle
- Ruler: http://www.cs.uu.nl/wiki/Ehc/Ruler

3.6 Exchanging Sources between Clean and Haskell

Report by:	Thomas van Noort
Participants:	John van Groningen, Peter Achten, Pieter Koopman, Rinus Plasmeijer
Status:	active development

In a Haskell'10 paper we describe how we facilitate the exchange of sources between Clean (→ 4.4) and Haskell. We use the existing Clean compiler as starting point, and implement a double-edged front end for this compiler: it supports both standard Clean 2.1 and (currently a large part of) standard Haskell 98. Moreover, it allows both languages to seamlessly use many of each other's language features that were alien to each other before. For instance, Haskell can now use uniqueness typing anywhere, and Clean can use newtypes efficiently. This has given birth to two new dialects of Clean and Haskell, dubbed Clean∗ and Haskell∗. Measurements of the performance of the new compiler indicate that it is on par with the flagship Haskell compiler GHC.

Future plans

Although the most important features of Haskell 98 have been implemented, the list of remaining issues is still rather long since some features took much more work than expected. Also, to enable the practical reuse of Haskell libraries, we have to implement some of GHC's extensions, such as generalised algebraic datatypes and type families. This is challenging, not only in terms of the programming effort, but more because of the consequences it will have on features such as uniqueness typing. We plan to use this double-edged front as an implementation laboratory to investigate these avenues.

Further reading

- John van Groningen, Thomas van Noort, Peter Achten, Pieter Koopman, and Rinus Plasmeijer. Exchanging sources between Clean and Haskell — A double-edged front end for the Clean compiler. In Jeremy Gibbons, editor, *Proceedings of the Haskell Symposium, Haskell '10, Baltimore, MD, USA*, pages 49–60. ACM Press, 2010.
- The front end is under active development, current releases are available via http://wiki.clean.cs.ru.nl/Download_Clean.

3.7 The Reduceron

Report by:	Matthew Naylor
Participants:	Colin Runciman, Jason Reich, Marco Perez Cervantes
Status:	experimental

The Reduceron is a graph-reduction processor implemented on an FPGA.

Over the past 18 months, work on the Reduceron has led to a factor of five speed-up. This has been achieved through a range of design improvements spanning architectural, machine, and compiler-level issues. See our ICFP'10 paper for details.

Work on the Reduceron continues. We have taken a step towards parallel reduction in the form of *primitive redex speculation*. We have developed a static analysis and transformation (currently limited to first-order programs) that predicts and increases run-time occurrences of primitive redexes, allowing a simpler and faster machine design. Early results look good, and we hope to extend the technique to higher-order programs.

Experiments in verification, both at the compiler level and the bytecode level, are also underway.

Looking ahead, we aim eventually to have multiple Reducerons running in parallel. We are also interested in increasing the amount of memory available to the Reduceron, and in technology advances that may enable faster clocking frequencies.

Two main by-products have emerged from the work. First, *York Lava*, now available from Hackage, is the HDL we use. It is very similar to Chalmers Lava (→ 11.6), but supports a greater variety of primitive components, behavioral description, number-parameterized types, and a first attempt at a Lava prelude. Second, *F-lite* is our subset of Haskell, with its own lightweight toolset and experimental supercompiler (http://haskell.org/communities/11-2009/html/report.html#sect4.1.4).

Further reading

- http://www.cs.york.ac.uk/fp/reduceron/
- http://hackage.haskell.org/package/york-lava/

3.8 Specific Platforms

3.8.1 Debian Haskell Group

Report by:	Joachim Breitner
Status:	working

The Debian Haskell Group aims to provide an optimal Haskell experience to users of the Debian GNU/Linux distribution and derived distributions such as Ubuntu. We try to follow the Haskell Platform versions for the core package and package a wide range of other useful libraries and programs. In total, we maintain 202 source packages.

A system of virtual package names and dependencies, based on the ABI hashes, guarantees that a system upgrade will leave all installed libraries usable. Most libraries are also optionally available with the profiling data and the documentation packages register with the system-wide index.

Currently, we are in the process of releasing the next version of Debian, squeeze, so the updating rate has slowed. Once this is done, we will bring our versions up to date. This will also require some work to rename the packages from libghc6- to libghc-, as the next version of GHC has a new major version number.

Further reading

http://wiki.debian.org/Haskell

3.8.2 Haskell in Gentoo Linux

Report by:	Lennart Kolmodin

Gentoo Linux currently officially supports GHC 6.10.4, including the latest Haskell Platform (→ 3.1) for x86, amd64, sparc, and ppc64. For previous GHC versions we also have binaries available for alpha, hppa and ia64.

The full list of packages available through the official repository can be viewed at http://packages.gentoo.org/category/dev-haskell?full_cat.

The GHC architecture/version matrix is available at http://packages.gentoo.org/package/dev-lang/ghc.

Please report problems in the normal Gentoo bug tracker at bugs.gentoo.org.

We have also recently started an official Gentoo Haskell blog where we can communicate with our users what we are doing http://gentoohaskell.wordpress.com/.

There is also an overlay which contains more than 300 extra unofficial and testing packages. Thanks to the Haskell developers using Cabal and Hackage (→ 6.7.1), we have been able to write a tool called "hackport" (initiated by Henning Günther) to generate Gentoo packages with minimal user intervention. Notable packages in the overlay include the latest version of the Haskell Platform as well as the latest 6.12.2 release of GHC, as well as popular Haskell packages such as pandoc (→ 9.2.3) and gitit (→ 5.2.5).

More information about the Gentoo Haskell Overlay can be found at http://haskell.org/haskellwiki/Gentoo. Using Darcs (→ 6.5.1), it is easy to keep up to date, to submit new packages, and to fix any problems in existing packages. It is also available via the Gentoo overlay manager "layman". If you choose to use the overlay, then any problems should be reported on IRC (`#gentoo-haskell` on freenode), where we coordinate development, or via email ⟨haskell@gentoo.org⟩ (as we have more people with the ability to fix the overlay packages that are contactable in the IRC channel than via the bug tracker).

Through recent efforts we have developed a tool called "haskell-updater" http://www.haskell.org/haskellwiki/Gentoo#haskell-updater (initiated by Ivan Lazar Miljenovic). This is a replacement of the old `ghc-updater` script for rebuilding packages when a new version of GHC is installed which is now not only written in Haskell but will also rebuild broken packages. "haskell-updater" is still in active development to further refine and add to its features and capabilities.

As always we are more than happy for (and in fact encourage) Gentoo users to get involved and help us maintain our tools and packages, even if it is as simple as reporting packages that do not always work or need updating: with such a wide range of GHC and package versions to co-ordinate, it is hard to keep up! Please contact us on IRC or email if you are interested!

3.8.3 Fedora Haskell SIG

Report by:	Jens Petersen
Participants:	Ben Boeckel, Shakthi Kannan, Lakshmi Narasimhan, Bryan O'Sullivan, Conrad Meyer, Fedora Haskell SIG
Status:	on-going

The Fedora Haskell SIG is an effort to provide good support for Haskell in Fedora.

Fedora 14 is shipping on 2nd November with ghc-6.12.3, haskell-platform-2010.2.0.0, and darcs-2.4.4. Library doc subpackages have been merged into their devel subpackages. Most of the new core gtk2hs packages have been packaged and xmobar was also added. There are currently 72 Haskell-related source packages in Fedora, and more than 60 new packages in the review queue.

In Fedora 15 we are hoping to ship ghc 7 and to use ghc package hash metadata in our binary rpms. Also more packages are planned: e.g., pandoc and leksah.

Contributions to Fedora Haskell are welcome: join us on #fedora-haskell on Freenode IRC and our mailing-list.

Further reading

- http://fedoraproject.org/wiki/SIGs/Haskell
- http://fedoraproject.org/wiki/Documentation_Development_Haskell_Beat

4 Related Languages

4.1 Agda

Report by:	Nils Anders Danielsson
Participants:	Ulf Norell, Andreas Abel, and many others
Status:	actively developed

Agda is a dependently typed functional programming language (developed using Haskell). A central feature of Agda is inductive families, i.e. GADTs which can be indexed by *values* and not just types. The language also supports coinductive types, parameterized modules, and mixfix operators, and comes with an *interactive* interface—the type checker can assist you in the development of your code.

A lot of work remains in order for Agda to become a full-fledged programming language (good libraries, mature compilers, documentation, etc.), but already in its current state it can provide lots of fun as a platform for experiments in dependently typed programming.

In September version 2.2.8 was released, with these new features:

◦ Pattern matching for records.

◦ Proof-irrelevant function types.

◦ Reflection.

◦ Users can define new forms of binding syntax.

Further reading

The Agda Wiki: http://wiki.portal.chalmers.se/agda/

4.2 MiniAgda

Report by:	Andreas Abel
Status:	experimental

MiniAgda is a tiny dependently-typed programming language in the style of Agda (→ 4.1). It serves as a laboratory to test potential additions to the language and type system of Agda. MiniAgda's termination checker is a fusion of sized types and size-change termination and supports coinduction. Equality incorporates eta-expansion at record and singleton types. Function arguments can be declared as static; such arguments are discarded during equality checking and compilation.

Currently, I am developing a source translation from MiniAgda to Haskell as prototype for an Agda to Haskell compiler. In the long run, I plan to evolve MiniAgda into a core language for Agda with termination certificates.

MiniAgda is available as Haskell source code and compiles with GHC 6.12.x.

Further reading

http://www2.tcs.ifi.lmu.de/~abel/miniagda/

4.3 Idris

Report by:	Edwin Brady
Status:	active development

Idris is an experimental language with full dependent types. Dependent types allow types to be predicated on values, meaning that some aspects of a program's behavior can be specified precisely in the type. The language is closely related to Epigram and Agda (→ 4.1). It is available from http://www.idris-lang.org, and there is a tutorial at http://www.idris-lang.org/tutorial.

Idris aims to provide a platform for realistic programming with dependent types. By realistic, we mean the ability to interact with the outside world and use primitive types and operations, to make a dependently typed language suitable for systems programming. This includes networking, file handling, concurrency, etc. Idris emphasizes programming over theorem proving, but nevertheless integrates with an interactive theorem prover. It is compiled, via C, and uses the Boehm-Demers-Weiser garbage collector.

One goal of the project is to show that Idris, and dependently typed programming in general, can be efficient enough for the development of real world verified software. To this end, Idris is currently being used to develop a library for verified network protocol implementation, with example applications.

Further reading

http://www.idris-lang.org/

4.4 Clean

Report by:	Thomas van Noort
Participants:	Rinus Plasmeijer, John van Groningen
Status:	active development

Clean is a general purpose, state-of-the-art, pure and lazy functional programming language designed for making real-world applications. Here is a short list of notable features:

◦ Clean is a lazy, pure, and higher-order functional programming language with explicit graph-rewriting semantics.

- Although Clean is by default a lazy language, one can smoothly turn it into a strict language to obtain optimal time/space behavior: functions can be defined lazy as well as (partially) strict in their arguments; any (recursive) data structure can be defined lazy as well as (partially) strict in any of its arguments.
- Clean is a strongly typed language based on an extension of the well-known Milner/Hindley/Mycroft type inferencing/checking scheme including the common higher-order types, polymorphic types, abstract types, algebraic types, type synonyms, and existentially quantified types.
- Clean has pattern matching, guards, list comprehensions, array comprehensions and a lay-out sensitive mode.
- Clean supports type classes and type constructor classes to make overloaded use of functions and operators possible.
- The uniqueness typing system in Clean makes it possible to develop efficient applications. In particular, it allows a refined control over the single-threaded use of objects which can influence the time and space behavior of programs. Uniqueness typing can also be used to incorporate destructive updates of objects within a pure functional framework. It allows destructive transformation of state information and enables efficient interfacing to the nonfunctional world (to C but also to I/O systems like X-Windows) offering direct access to file systems and operating systems.
- Clean offers records and (destructively updateable) arrays and files.
- The Clean type system supports dynamic typing, allowing values of arbitrary types to be wrapped in a uniform package and unwrapped via a type annotation at run time. Using dynamics, code and data can be exchanged between Clean applications in a flexible and type-safe way.
- Clean provides a built-in mechanism for generic functions.
- There is a Clean IDE and there are many libraries available offering additional functionality.

Future plans

Please see the entry on exchanging sources between Clean and Haskell (→ 3.6) for the future plans.

Further reading

http://wiki.clean.cs.ru.nl/

4.5 Timber

Report by:	Johan Nordlander
Participants:	Björn von Sydow, Andy Gill, Magnus Carlsson, Per Lindgren, Thomas Hallgren, and others
Status:	actively developed

Timber is a general programming language derived from Haskell, with the specific aim of supporting development of complex event-driven systems. It allows programs to be conveniently structured in terms of objects and reactions, and the real-time behavior of reactions can furthermore be precisely controlled via platform-independent timing constraints. This property makes Timber particularly suited to both the specification and the implementation of real-time embedded systems. An implementation of Timber is available as a command-line compiler tool, currently targeting POSIX-based systems only.

Timber shares most of Haskell's syntax but introduces new primitive constructs for defining classes of reactive objects and their methods. These constructs live in the *Cmd* monad, which is a replacement of Haskell's top-level monad offering mutable encapsulated state, implicit concurrency with automatic mutual exclusion, synchronous as well as asynchronous communication, and deadline-based scheduling. In addition, the Timber type system supports nominal subtyping between records as well as datatypes, in the style of its precursor O'Haskell.

A particularly notable difference between Haskell and Timber is that Timber uses a *strict* evaluation order. This choice has primarily been motivated by a desire to facilitate more predictable execution times, but it also brings Timber closer to the efficiency of traditional execution models. Still, Timber retains the purely functional characteristic of Haskell, and also supports construction of recursive structures of arbitrary type in a declarative way.

The Timber compiler is currently undergoing a major reimplementation of its front-end, an effort triggered by increasing needs to significantly improve error messages as well as to sharpen up the documentation of the language syntax and its scoping rules. The new compiler, tentatively called version 2, will also include a newly developed Javascript back-end, HTML5 and OpenGL bindings, as well as bare-metal ARM7 support. A minor bug-fix release announced in the previous HCAR has been postponed and will be merged into the release of version 2. The latest release of the Timber compiler system still dates back to May 2009 (version 1.0.3).

Other active projects include interfacing the compiler to memory and execution-time analysis tools, extending it with a supercompilation pass, and building an interpreting debugger on basis of the new compiler front-end.

Further reading

http:://timber-lang.org

4.6 Disciple

Report by:	Ben Lippmeier
Participants:	Erik de Castro Lopo
Status:	experimental, active development

Disciple is a dialect of Haskell that uses strict evaluation as the default and supports destructive update of arbitrary data. Many Haskell programs are also Disciple programs, or will run with minor changes. In addition, Disciple includes region, effect, and closure typing, and this extra information provides a handle on the operational behaviour of code that is not available in other languages. Our target applications are the ones that you always find yourself writing C programs for, because existing functional languages are too slow, use too much memory, or do not let you update the data that you need to.

Our compiler (DDC) is still in the "research prototype" stage, meaning that it will compile programs if you are nice to it, but expect compiler panics and missing features. You will get panics due to ungraceful handling of errors in the source code, but valid programs should compile ok. The test suite includes a few thousand-line graphical demos, like a ray-tracer and an n-body collision simulation, so it is definitely hackable.

We have spent a good slab of time this year cleaning up the internals and getting proper regression testing build bots online. We now support OSX/x86, Linux/{x86, x86–64, PPC}, FreeBSD/x86, and Cygwin/x86 so you should be able to get DDC running on your own system without trouble. Other than that, we have been stabilising the existing implementation and fixing bugs. The plan for the coming year is to complete support for type classes and dictionary passing, and to extend the type system so that it can "auto-freeze" data structures that have been created using destructive update but will be treated as constant from then on. We are also working on an LLVM port which will provide faster code in the long term without having to rely on the existing via-C backend.

Disciple programs can be written in either a pure/functional or effectful/imperative style, and one of our main goals is to provide both styles coherently in the same language. The two styles can be mixed safely. For example: when using laziness, the type system guarantees that computations with visible side effects are not suspended. The fact that we have region, effect, and closure typing available means we can also support more fine-grained notions of ST-monad style effect encapsulation, with the added benefit that the encapsulation/masking is handled seamlessly by the type system. If this sounds interesting to you then drop us a line!

Further reading

http://trac.haskell.org/ddc

5 Haskell and . . .

5.1 Haskell and Parallelism

5.1.1 TwilightSTM

Report by:	Annette Bieniusa
Participants:	Arie Middelkoop, Peter Thiemann
Status:	experimental

TwilightSTM is an extended Software Transactional Memory system. It safely augments the STM monad with non-reversible actions and allows introspection and modification of a transaction's state.

TwilightSTM splits the code of a transaction into a (functional) atomic phase, which behaves as in GHC's implementation, and an (imperative) twilight phase. Code in the twilight phase executes before the decision about a transaction's fate (restart or commit) is made and can affect its outcome based on the actual state of the execution environment.

The Twilight API has operations to detect and repair read inconsistencies as well as operations to overwrite previously written variables. It also permits the safe embedding of I/O operations with the guarantee that each I/O operation is executed only once. In contrast to other implementations of irrevocable transactions, twilight code may run concurrently with other transactions including their twilight code in a safe way. However, the programmer is obliged to prevent deadlocks and race conditions when integrating I/O operations that participate in locking schemes.

A prototype implementation is available on Hackage (http://hackage.haskell.org/package/twilight-stm). We are currently working on the composability of Twilight monads and are applying TwilightSTM to different use cases.

Further reading

http://proglang.informatik.uni-freiburg.de/projects/twilight/

5.1.2 Haskell-MPI

Report by:	Bernie Pope
Participants:	Dmitry Astapov, Duncan Coutts
Status:	first public version to be released soon

MPI, the *Message Passing Interface*, is a popular communications protocol for distributed parallel computing (http://www.mpi-forum.org/). It is widely used in high performance scientific computing, and is designed to scale up from small multi-core personal computers to massively parallel supercomputers. MPI applications consist of independent computing processes which share information by message passing communication. It supports both point-to-point and collective communication operators, and manages much of the mundane aspects of message delivery. There are several high-quality implementations of MPI available which adhere to the standard API specification (the latest version of which is 2.2). The MPI specification defines interfaces for C, C++, and Fortran, and bindings are available for many other programming languages. As the name suggests, Haskell-MPI provides a Haskell interface to MPI, and thus facilitates distributed parallel programming in Haskell. It is implemented on top of the C API via Haskell's foreign function interface. Haskell-MPI provides three different ways to access MPI's functionality:

1. A direct binding to the C interface.
2. A convenient interface for sending arbitrary serializable Haskell data values as messages.
3. A high-performance interface for working with (possibly mutable) arrays of storable Haskell data types.

We do not currently provide exhaustive coverage of all the functions and types defined by MPI 2.2, although we do provide bindings to the most commonly used parts. In the future we plan to extend coverage based on the needs of projects which use the library.

We are in the final stages of preparing the first release of Haskell-MPI. We will publish the code on Hackage once the user documentation is complete. We have run various simple latency and bandwidth tests using up to 512 Intel x86-64 cores, and for the high-performance interface, the results are within acceptable bounds of those achieved by C. Haskell-MPI is designed to work with any compliant implementation of MPI, and we have successfully tested it with both OpenMPI (http://www.open-mpi.org/) and MPICH2 (http://www.mcs.anl.gov/research/projects/mpich2/).

Further reading

http://github.com/bjpop/haskell-mpi

5.1.3 Eden

Report by:	Rita Loogen
Participants:	**in Madrid:** Yolanda Ortega-Mallén, Mercedes Hidalgo, Lidia Sánchez-Gil, Fernando Rubio, Alberto de la Encina, **in Marburg:** Mischa Dieterle, Thomas Horstmeyer, Oleg Lobachev, Rita Loogen, Bernhard Pickenbrock **in Copenhagen:** Jost Berthold
Status:	ongoing

Eden extends Haskell with 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), synchronization, and process handling.

Eden's main constructs are process abstractions and process instantiations. The function `process :: (a -> b) -> Process a b` embeds a function of type `(a -> b)` into a *process abstraction* of type `Process a b` which, when instantiated, will be executed in parallel. *Process instantiation* is expressed by the predefined infix operator `( # ) :: Process a b -> a -> b`. Higher-level coordination is achieved by defining *skeletons*, ranging from a simple parallel map to sophisticated replicated-worker schemes. They have been used to parallelize a set of non-trivial benchmark programs.

Survey and standard reference

Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña: *Parallel Functional Programming in Eden*, Journal of Functional Programming 15(3), 2005, pages 431–475.

Implementation

We happily announce that a new release of the Eden compiler based on GHC 6.12.3 is available on our relaunched web pages, see

http://www.mathematik.uni-marburg.de/~eden

New features are the support of 64-Bit architectures and an extended version of the new GHC EventLog format for parallel program traces. These traces can be visualized using the Eden trace viewer tool EdenTV. The new version of this tool has been written in Haskell and is also freely available on the Eden web pages.

The Eden skeleton library is currently being revised and cabalized. A development snapshot is available on the Eden pages.

Recent and Forthcoming Publications

- Lidia Sánchez-Gil, Mercedes Hidalgo-Herrero, Yolanda Ortega-Mallén: *On the relation of call-by-need and call-by-name in a natural semantics setting*, In Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010), Technical Report UU-CS-2010-020, Department of Information and Computing Sciences, Utrecht University, 2010.
- Jost Berthold: *Orthogonal Haskell Data Serialisation*, In Preproceedings of the 22nd Symposium on Implementation and Application of Functional Languages (IFL 2010), Technical Report UU-CS-2010-020, Department of Information and Computing Sciences, Utrecht University, 2010.
- Oleg Lobachev and Rita Loogen: *Estimating Parallel Performance, a Skeleton-based Approach*, in HLPP'10: Workshop on High-level Parallel Programming, ACM Press, 2010, 25–34.
- Oleg Lobachev, Rita Loogen: *Implementing Data Parallel Rational Multiple-Residue Arithmetic in Eden*, in CASC'10: Computer Algebra in Scientific Computing, Springer LNCS 6244, 2010, 178–193.
- Mischa Dieterle, Jost Berthold, Rita Loogen: *A Skeleton for Distributed Work Pools in Eden*, in FLOPS'10: Functional and Logic Programming, Springer LNCS 6009, 2010, 337-353.
- Thomas Horstmeyer, Rita Loogen: *Grace — Graph-based Communication in Eden*, Trends in Functional Programming, Volume 10, Intellect 2010, 1–16.
- Mustafa Aswad, Phil Trinder, Abdallah Al Zain, Greg Michaelson, Jost Berthold: *Low Pain vs No Pain Multi-core Haskells*, Trends in Functional Programming, Volume 10, Intellect 2010, 49–64.
- Lidia Sánchez-Gil, Mercedes Hidalgo-Herrero, Yolanda Ortega-Mallén: *An Operational Semantics for Distributed Lazy Evaluation*, Trends in Functional Programming, Volume 10, Intellect 2010, 65–80.

Further reading

http://www.mathematik.uni-marburg.de/~eden

5.2 Haskell and the Web

5.2.1 GHCJS: Haskell to Javascript compiler

Report by:	Victor Nazarov
Status:	0.1.0 released

GHCJS currently is a GHC back-end which produces Javascript code. Modern Javascript environments become more and more advanced. TraceMonkey and V8 engines allow very fast Javascript execution. It is possible, for instance, to create an in-browser hardware emulator: an emulated CPU's instructions are compiled down to Javascript functions, and Javascript instructions are compiled to the native host CPU's instructions by Javascript JIT-compilers (http://weblogs.mozillazine.org/roc/archives/2010/11/implementing_a.html).

The idea to bring the power of the Haskell language to the world of AJAX-applications is not new. It has been proposed many times in Haskell-café. The success of Google's GWT was uncomfortable to watch, when our beloved language lacked such a feature. The first implementation I know is Dmitry Golubovsky's YHC back-end (http://www.haskell.org/haskellwiki/Yhc/Javascript). The second one was my GHC backend hs2js (http://vir.mskhug.ru/). There were differences between the two projects. Dmitry had tried to provide a Haskell environment to develop everything in Haskell. He had developed an automated conversion tool to generate Haskell-bindings from DOM IDL specifications provided by the W3C. My aim was more modest: I thought that we could use Haskell to implement complex logic. The ability to use Parsec in a browser was asked for several times in Haskell-café. With the latter approach we can extend existing Javascript-applications with algorithms implemented in Haskell. UHC (→ 3.5) started to implement a Javascript-backend recently (http://utrechthaskellcompiler.wordpress.com/2010/10/18/haskell-to-javascript-backend/), but I have not looked at it, yet.

GHCJS is a fresh rewrite of hs2js that was started in August 2010. It is currently a standalone tool that uses GHC as a library and produces a `.js`-file for each Haskell-module. Javascript code can load any Haskell-module and evaluate any exported Haskell-value. Some examples that are available with the GHCJS package show some simple Haskell programs like generation of a sequence of prime-numbers. Each Haskell module is currently a standalone Javascript file. When a value of some module is needed, the module is loaded dynamically.

The code is available at the GHCJS github page (see below) under the terms of the BSD3 license. It was tested with GHC 6.12.

There are many tasks awaiting completion with GHCJS:

A faster and more robust module loader: Now it loses a lot of time on 404 errors, trying to access modules in the wrong package directory. I plan to use GHC's package abstraction. A package will be a Web-server's directory and Javascript's namespace. Every module will be unambiguously associated with one package. It will become possible to load a module with one unambiguous HTTP-request. This change will short the loading time of Haskell programs.

Make it work in all major browsers: There are some minor problems with Internet Explorer. But it should be trivial to fix them.

FFI support: FFI support should make the whole thing generally usable. FFI-exports should generate easily-callable Javascript functions that will type-check their arguments to make a combination of dynamically-typed Javascript and statically-typed Haskell seamless. FFI-imports will allow the implementation of DOM-manipulation in Haskell programs.

Further reading

https://github.com/sviperll/ghcjs

5.2.2 Hawk

Report by:	Uwe Schmidt
Participants:	Björn Peemöller, Stefan Roggensack, Alexander Treptow
Status:	first release

The Hawk system is a web framework for Haskell. It is comparable in functionality and architecture with Ruby on Rail and other web frameworks. Its architecture follows the MVC pattern. It consists of a simple relational database mapper for persistent storage of data and a template system for the view component. This template system has two interesting features: First, the templates are valid XHTML documents. The parts where data has to be filled in are marked with Hawk specific elements and attributes. These parts are in a different namespace, so they do not destroy the XHTML structure. The second interesting feature is that the templates contain type descriptions for the values to be filled in. This type information enables a static type check whether the models and views fit together.

A first application of the Hawk framework is a customizable search for Hayoo! (→ 5.2.4). But the framework is independent of the Holumbus search engine. It will be applicable for the development of arbitrary web applications.

Hawk was developed by Björn Peemöller and Stefan Roggensack. Currently, Alexander Treptow is applying, testing, and extending the framework.

5.2.3 WAI

Report by:	Michael Snoyman
Status:	stable

The Web Application Interface (WAI) is an interface between web applications and web servers. By targeting the WAI, a web application can get access to multiple servers; and through WAI, a server can support web applications never intended to run on it.

In designing this package, performance was first priority: there should be no performance overhead for using the WAI. As such, an enumerator interface was selected for the response body, a handle-like interface, called a source, for the request body, and bytestrings used throughout. Another design decision was to keep the interface as general as possible by excluding variables which are not universal to all web servers.

Since the last report, version 0.2.0 has been released, which replaces some of the special data types (request and response headers, for instance) with CIByteString, a case-insensitive bytestring which allows easy lookups. The ecosystem around WAI has also matured significantly: we have handlers for CGI, FastCGI, SCGI, development servers and the Snap standalone server (→ 5.2.10), and middleware for cleaning URLs, GZIP compression, and JSON-P. There is even a backend to convert your web applications into desktop applications via Webkit.

Hopefully, WAI can be one of many smaller packages which lead to collaboration in the Haskell web development community and development of a healthy ecosystem. There is an experimental Happstack WAI backend, and the Yesod Web Framework (→ 5.2.8) uses WAI exclusively.

Further reading

http://github.com/snoyberg/wai

5.2.4 Holumbus Search Engine Framework

Report by:	Uwe Schmidt
Participants:	Timo B. Hübel, Sebastian Gauck, Stefan Schmidt, Björn Peemöller, Stefan Roggensack, Sebastian Reese, Alexander Treptow
Status:	first release

Description

The Holumbus framework consists of a set of modules and tools for creating fast, flexible, and highly customizable search engines with Haskell. The framework consists of two main parts. The first part is the indexer for extracting the data of a given type of documents, e.g., documents of a web site, and store it in an appropriate index. The second part is the search engine for querying the index.

An instance of the Holumbus framework is the Haskell API search engine Hayoo! (http://holumbus.fh-wedel.de/hayoo/). The web interface for Hayoo! is implemented with the Janus web server, written in Haskell and based on HXT (→ 8.8.2).

The framework supports distributed computations for building indexes and searching indexes. This is done with a MapReduce like framework. The MapReduce framework is independent of the index- and search-components, so it can be used to develop distributed systems with Haskell.

The framework is now separated into four packages, all available on Hackage.

- The Holumbus Search Engine
- The Holumbus Distribution Library
- The Holumbus Storage System
- The Holumbus MapReduce Framework

The search engine package includes the indexer and search modules, the MapReduce package bundles the distributed MapReduce system. This is based on two other packages, which may be useful for their on: The Distributed Library with a message passing communication layer and a distributed storage system.

Features

- Highly configurable crawler module for flexible indexing of structured data
- Customizable index structure for an effective search
- *find as you type* search
- Suggestions
- Fuzzy queries
- Customizable result ranking
- Index structure designed for distributed search
- Git repository containing the current development version of all packages under http://holumbus.fh-wedel.de/src.git
- Distributed building of search indexes

Current Work

The data structures of the Holumbus indexes have been optimized for space and time. There is a new and efficient prefix tree structure, which further enables index updates.

The indexer and search module is used to support the Hayoo! engine for searching the hackage package library (http://holumbus.fh-wedel.de/hayoo/hayoo.html). Because of the fast growing number of packages on hackage, the Hayoo! search engine will be extended by a package search.

Sebastian Reese has finished his work on applying the MapReduce framework and for giving tuning and configuration hints. Benchmarks for various small problems and for generating search indexes have shown that the architecture scales very well.

In a subproject of Holumbus, the so called Hawk framework (→ 5.2.2), Björn Peemöller and Stefan Roggensack have developed a web framework for Haskell. Currently Alexander Treptow is applying, testing, and extending the framework. A first application is a customizable search for Hayoo!

Further reading

The Holumbus web page (http://holumbus.fh-wedel.de/) includes downloads, Git web interface, current status, requirements, and documentation. Timo Hübel's master thesis describing the Holumbus index structure and the search engine is available at http://holumbus.fh-wedel.de/branches/develop/doc/thesis-searching.pdf. Sebastian Gauck's thesis dealing with the crawler component is available at http://holumbus.fh-wedel.de/src/doc/thesis-indexing.pdf The thesis of Stefan Schmidt describing the Holumbus MapReduce is available via http://holumbus.fh-wedel.de/src/doc/thesis-mapreduce.pdf.

5.2.5 gitit

Report by:	John MacFarlane
Participants:	Gwern Branwen, Simon Michael, Henry Laxen, Anton van Straaten, Robin Green, Thomas Hartman, Justin Bogner, Kohei Ozaki, Dmitry Golubovsky, Anton Tayanovskyy, Dan Cook, Jinjing Wang
Status:	active development

Gitit is a wiki built on Happstack (→ 5.2.6) and backed by a git, darcs, or mercurial filestore. Pages and uploaded files can be modified either directly via the VCS's command-line tools or through the wiki's web interface. Pandoc (→ 9.2.3) is used for markup processing, so pages may be written in (extended) markdown, reStructuredText, LaTeX, HTML, or literate Haskell, and exported in thirteen different formats, including LaTeX, ConTeXt, DocBook, RTF, OpenOffice ODT, MediaWiki markup, EPUB, and PDF.

Notable features of gitit include:

- Plugins: users can write their own dynamically loaded page transformations, which operate directly on the abstract syntax tree.
- Math support: LaTeX inline and display math is automatically converted to MathML, using the `texmath` library.
- Highlighting: Any git, darcs, or mercurial repository can be made a gitit wiki. Directories can be browsed, and source code files are automatically syntax-highlighted. Code snippets in wiki pages can also be highlighted.
- Library: Gitit now exports a library, `Network.Gitit`, that makes it easy to include a gitit wiki (or wikis) in any Happstack application.
- Literate Haskell: Pages can be written directly in literate Haskell.

Further reading

http://gitit.net (itself a running demo of gitit)

5.2.6 Happstack

Report by:	Jeremy Shaw

Happstack is a web application framework focused on high-scalability, rapid development, ease of deployment, and flexibility. The core libraries provided by Happstack include:

happstack-server , an HTTP server with a rich environment for routing requests, working with cookies, processing form data, handling file uploads, serving static content with `sendfile()`, and more. Applications can be run using the built-in HTTP backend, or by using other handlers such as FastCGI.

happstack-state , also known as MACID, provides a NoSQL, RAM-cloud for distributed persistent state. Unlike limited key-value stores, MACID natively stores arbitrary Haskell data types. This allows the storage of user defined types as well as standard data structures including trees, graphs, and maps. Updates and queries are written using plain old Haskell functions. These features are provided without sacrificing the ACID properties.

happstack-ixset provides a set data-type with the ability to index elements by multiple keys. It provides much of the same functionality as a table in a relational database. This includes the ability to search by one or more keys, search by range, update the value at a specified index, etc.

happstack-data builds on the binary library to provide versioned data serialization and automatic migration of data from older versions to newer versions.

Happstack also has integrated support for many web related libraries including:

templates using HSP, Hamlet, HStringTemplate, BlazeHtml, and more.

type-safe urls and routing avoid bad links and namespace collisions using web-routes.

form generation and validation using formlets or digestive functors.

databases using HDBC, Takusen, HaskellDB, etc.

Future plans

Happstack 6 is nearly completed. Happstack 6 features many improvements and performance enhancements to the happstack-server library. It has been heavily refactored to make documentation browsing easier. The haddock documentation has been greatly improved. And there is now a detailed Happstack Crash Course which guides developers through the libraries in a detailed and logical manner. It includes many self-contained runnable demos.

Happstack 7 will include significant enhancements to the happstack-state library including sharding and better tools for examining and manipulating the contents of the data store. It will also include a new implementation of happstack-ixset which is faster, uses less memory, and has support for parallel traversals to take advantage of multicore machines.

Happstack 8 will migrate to an iteratee-based HTTP backend for even better performance and resource management. The tentative plan is to use Hyena.

Further reading

- http://www.happstack.com/
- http://www.happstack.com/docs/crashcourse/index.html

5.2.7 Mighttpd — Yet another Web Server

Report by:	Kazu Yamamoto
Status:	open source, actively developed

Mighttpd (called mighty) is a simple but practical Web server in Haskell. It is now working on Mew.org providing basic web features and CGI (mailman and contents search). Three packages are registered in hackageDB.

c10k Since GHC is using the select system call, a Haskell program complied with GHC cannot handle over 1,024 connections/files simultaneously. The c10k package uses the prefork technique to get rid of this barrier.

webserver The webserver package provides HTTP parser, session management, redirection, CGI, and so on. This package is independent from back-end storage systems. So you can build a Web server on any storage system including files, key-value-store DB, etc.

mighttpd This package provides a simple but practical web server based on files using the c10k and webserver packages.

I am planning to implement FastCGI and WebSocket.

Further reading

http://www.mew.org/~kazu/proj/mighttpd/en/

5.2.8 Yesod

Report by:	Michael Snoyman
Status:	experimental

Yesod is a web framework designed to play towards the strengths of the Haskell language to make web programming safer and more productive. It is fair to say that most web development today occurs in dynamic languages like PHP, Python, and Ruby, and we see the results: cross-site scripting attacks, applications that do not scale, and countless minor bugs entering production because they can only be detected at runtime.

Instead of providing a single monolithic package, Yesod is broken up into many smaller projects. This means that many of the powerful features of Yesod can be used in your own web development tool stack without issue. Packages for authentication, client-side encrypted session data, middlewares, web encodings, YAML, persistence, HTML templating and more are all fully available on Hackage, without any reliance on Yesod.

Yesod is currently on its 0.5 version. There are plans for some minor changes to take place in 0.6: mostly this involves extracting some functionality into separate packages for more flexibility in API changes. Assuming this change goes well, 0.6 will probably morph into a 1.0 release, indicating a fair level of API stability.

The Yesod documentation site (http://docs.yesodweb.com/) is a great place for information. It has code examples, screencasts, the Yesod blog and—most importantly—a book on Yesod. The book is not yet complete, but provides a very solid introduction to the main features, and it is constantly being revised and expanded.

Yesod is already powering some major sites, including Haskellers (→ 1.1). This not only shows that Yesod is ready for use today, but also gives some great examples of real-life Yesod code in the wild. If you are looking for type-safe, concise, RESTful web development, you should check out Yesod.

Further reading

http://docs.yesodweb.com/

5.2.9 Lemmachine

Report by:	Larry Diehl
Participants:	Jason Dusek
Status:	experimental, active development

Lemmachine is a REST'ful web framework that makes it easy to get HTTP right by exposing users to overridable hooks with sane defaults. The main architecture is a copy of Erlang-based Webmachine, which is currently the best documentation reference (for hooks & general design).

Lemmachine stands out from the dynamically typed Webmachine by being written in dependently typed Agda (→ 4.1). The goal of the project is to show the advantages gained from compositional testing by taking advantage of proofs being inherently compositional. See http://github.com/larrytheliquid/Lemmachine/blob/master/src/Lemmachine/Default/Proofs.agda for examples of universally quantified proofs (tests over all possible input values) written against the default resource, which does not override any hooks.

When a user implements their own resource, they can write simple lemmas ("unit tests") against the resource's hooks, but then literally reuse those lemmas to write more complex proofs ("integration tests"). For examples see some reuse of lemmas in the proofs.

The big goal is to show that in service oriented architectures, proofs of individual middlewares can themselves be reused to write cross-service proofs (even higher level “integration tests”) for a consumer application that mounts those middlewares. See a post at http://vision-media.ca/resources/ruby/ruby-rack-middleware-tutorial for what is meant by middleware.

Another goal is for Lemmachine to come with proofs against the default resource (as it already does). Any hooks the user does not override can be given to the user for free by the framework! Anything that is overridden can generate proofs parameterized only by the extra information the user would need to provide. This would be a major boost in productivity compared to traditional languages whose libraries cannot come with tests for the user that have language-level semantics for real proposition reuse!

Lemmachine currently uses the Haskell Hack abstraction so it can run on several Haskell webservers. Because Agda compiles to Haskell and has an FFI, existing Haskell code can be integrated quite easily.

The project is still in development and rapidly changing. Lemmas and proofs exist for status resolution, and you can now run resources! The focus will now comprise of a gradual direct translation of RFC 2616 sections into dependent type theory.

Further reading

http://github.com/larrytheliquid/Lemmachine

5.2.10 Snap Framework

Report by:	Doug Beardsley
Participants:	Gregory Collins, Shu-yu Guo, James Sanders, Carl Howells, Shane O’Brien
Status:	active development

The Snap Framework is a web application framework built from the ground up for speed, reliability, and ease of use. The project’s goal is to be a cohesive high-level platform for web development that leverages the power and expressiveness of Haskell to make building websites quick and easy.

The Snap Framework has been quite active since the last HCAR. Several developers have joined the effort, the codebase has matured, and test coverage has increased. Recent benchmarks using the upcoming GHC 7 show approximately a 50% speed improvement from the benchmarks posted when we launched the project back in May. These speed improvements come as a result of improvements to both GHC and Snap.

The team is currently working on the upcoming 0.3 release which will include a more flexible library interface and support for automatic recompilation of apps amongst other things.

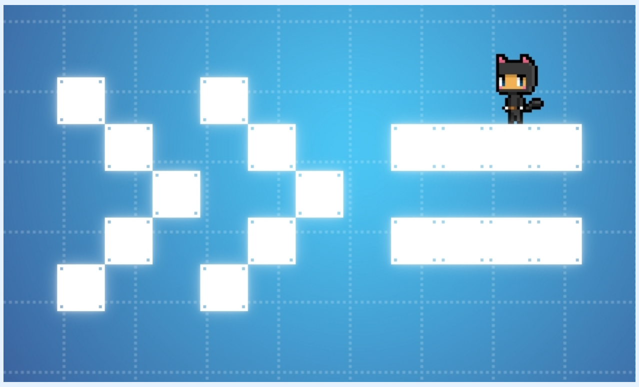
Further reading

http://snapframework.com

5.3 Haskell and Games

5.3.1 Nikki and the Robots

Report by:	Sönke Hahn
Participants:	Joyride Laboratories GbR
Status:	alpha, active



Nikki and the Robots is a 2D platformer written in Haskell and produced by Joyride Laboratories. Nikki, the protagonist, walks and jumps around the levels wearing a cute ninja/cat costume. Nikki refrains from using any tools or weapons, with one exception: The Robots. These come in various types with different abilities and can be used by Nikki to solve puzzles, overcome obstacles, and complete the level tasks. The game will feature an integrated level editor.

Publishing

We are releasing the game and the level editor under an open source license (LGPL). The included graphics are published under a permissive Creative Commons license (cc-by-sa). We are also planning to create a server that will allow players to upload the levels they created and download levels from other players. We hope that a community of coders, level creators, and players will emerge around the game.

Simultaneously, we are working on episodes that we plan to sell via the game. These will include new graphics, more robots, a story line, other characters, and other surprises.

(Just to clarify: The licensing is very permissive. It allows others to create their own episodes and distribute them freely or sell them. This would be very welcome. If anybody is interested in this, we propose to join forces and sell all our episodes through one system.)

Technologies Used

- Qt for user input and rendering.
- OpenGL as an efficient rendering backend for Qt. Everything will remain 2D, though - we promise!

- Hipmunk, the Haskell bindings to the chipmunk physics engine.

Getting Involved

The project is still in alpha stage, so there are some features that are not yet implemented. For some, we have a clear vision on how to implement them; for others, we do not. If you want to get involved, check out our darcs repo, our launchpad site, and do not hesitate to contact us.

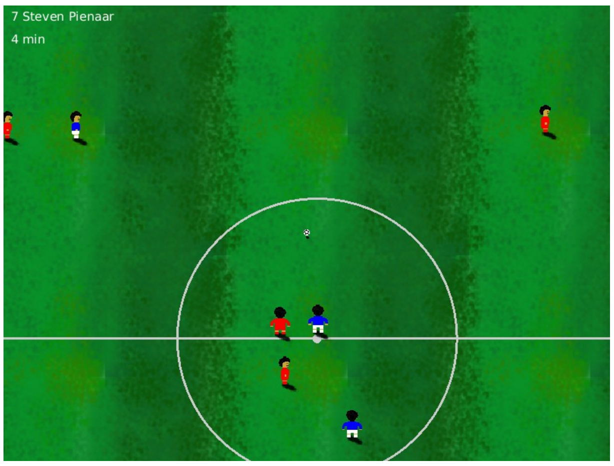
Further reading

- http://joyridelabs.de
- http://joyridelabs.de/game/code/

5.3.2 Freekick2

Report by:	Antti Salonen
Status:	experimental, active development

Freekick2 is a 2D arcade-style soccer game, written in Haskell. It is still very young, but playable. It features texture-mapped graphics, a simple but functional and well playing AI, and the ability to import Sensible Soccer team data files. Freekick2 uses the Haskell bindings to OpenGL, FTGL and SDL for input handling, graphics and GUI. It is available at Hackage. Future plans include improving the AI and the gameplay.



Further reading

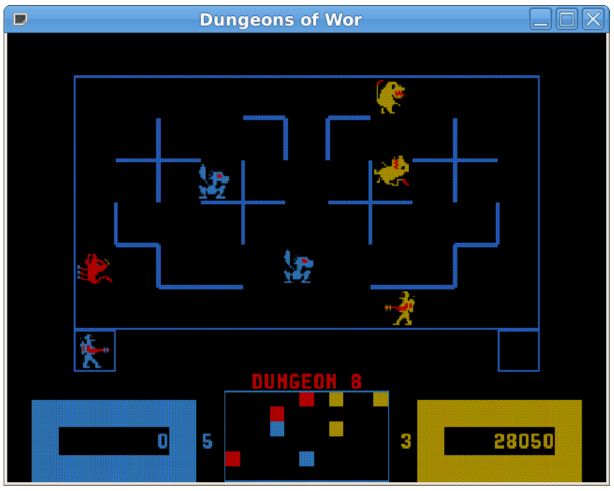
- http://github.com/anttisalonen/freekick2
- http://codeflow.wordpress.com/2010/05/04/announcing-freekick2/

5.3.3 Dungeons of Wor

Report by:	Patai Gergely
Status:	experimental, active

Dungeons of Wor is an homage to the classic arcade game, Wizard of Wor. It uses the artwork and levels from the arcade version, but the gameplay mechanics differ from the original in several ways.

This game is also an experiment in functional reactive programming, so it might be a useful resource to anyone interested in this topic. It was coded using the Simple variant of the experimental Elerea library (→ 9.3.2), which provides discrete streams as first-class values.



Further reading

- http://hackage.haskell.org/package/dow
- http://en.wikipedia.org/wiki/Wizard_of_Wor

5.4 Haskell and Compiler Writing

5.4.1 UUAG

Report by:	Arie Middelkoop
Participants:	ST Group of Utrecht University
Status:	stable, maintained

UUAG is the *Utrecht University Attribute Grammar* system. It is a preprocessor for Haskell which makes it easy to write *catamorphisms* (i.e., functions that do to any data type what *foldr* does to lists). You define tree walks using the intuitive concepts of *inherited* and *synthesized attributes*, while keeping the full expressive power of Haskell. The generated tree walks are *efficient* in both space and time.

An AG program is a collection of rules, which are pure Haskell functions between attributes. Idiomatic tree computations are neatly expressed in terms of copy, default, and collection rules. Attributes themselves can masquerade as subtrees and be analyzed accordingly (higher-order attribute). The order in which to visit the tree is derived automatically from the attribute computations. The tree walk is a single traversal from the perspective of the programmer.

Nonterminals (data types), productions (data constructors), attributes, and rules for attributes can be specified separately, and are woven and ordered automatically. These aspect-oriented programming features make AGs convenient to use in large projects.

The system is in use by a variety of large and small projects, such as the Utrecht Haskell Compiler UHC (→ 3.5), the editor Proxima for structured documents (→ 9.2.6), the Helium compiler (→ 3.4), the Generic Haskell compiler, UUAG itself, and many master student projects. The current version is 0.9.29 (July 2010), is extensively tested, and is available on Hackage.

We are working on the following enhancements of the UUAG system:

First-class AGs We provide a translation from UUAG to AspectAG (→ 5.4.2). AspectAG is a library of strongly typed Attribute Grammars implemented using type-level programming. With this extension, we can write the main part of an AG conveniently with UUAG, and use AspectAG for (dynamic) extensions. Our goal is to have an extensible version of the UHC.

Fixpoint evaluation We incorporated a fixed-point evaluation scheme for circular grammars. A cycle is broken by specifying an initial value for an attribute on the cycle, and repeating the evaluation with an updated value until it converges.

Step-wise evaluation We provide the possibility to evaluate AGs step-wise. The evaluation for a nonterminal may yield user-defined progress reports, and we can direct the evaluation until the next progress report. With this mechanism, we can resolve nondeterminism and encode breadth-first search strategies.

Further reading

- http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarSystem
- http://hackage.haskell.org/package/uuagc

5.4.2 AspectAG

Report by:	Marcos Viera
Participants:	Doaitse Swierstra, Wouter Swierstra
Status:	experimental

AspectAG is a library of strongly typed Attribute Grammars implemented using type-level programming.

Introduction

Attribute Grammars (AGs), a general-purpose formalism for describing recursive computations over data types, avoid the trade-off which arises when building software incrementally: should it be easy to add new data types and data type alternatives or to add new operations on existing data types? However, AGs are usually implemented as a pre-processor, leaving e.g. type checking to later processing phases and making interactive development, proper error reporting and debugging difficult. Embedding AG into Haskell as a combinator library solves these problems. Previous attempts at embedding AGs as a domain-specific language were based on extensible records and thus exploiting Haskell's type system to check the well-formedness of the AG, but fell short in compactness and the possibility to abstract over oft occurring AG patterns. Other attempts used a very generic mapping for which the AG well-formedness could not be statically checked. We present a typed embedding of AG in Haskell satisfying all these requirements. The key lies in using HList-like typed heterogeneous collections (extensible polymorphic records) and expressing AG well-formedness conditions as type-level predicates (i.e., typeclass constraints). By further type-level programming we can also express common programming patterns, corresponding to the typical use cases of monads such as Reader, Writer, and State. The paper presents a realistic example of type-class-based type-level programming in Haskell.

Current Status

In the current version (0.3) we have included support for local and higher-order attributes. Furthermore, a translation from UUAG (→ 5.4.1) to AspectAG is added to UUAGC as an experimental feature.

Background

The approach taken in AspectAG was proposed by Marcos Viera, Doaitse Swierstra, and Wouter Swierstra in the ICFP 2009 paper "Attribute Grammars Fly First-Class: How to do aspect oriented programming in Haskell".

Further reading

http://www.cs.uu.nl/wiki/bin/view/Center/AspectAG

5.4.3 Berp

Report by:	Bernie Pope
Status:	under development

Berp is an implementation of Python 3. At its heart it is a translator which takes Python code as input and generates Haskell code as output. The Haskell code is fed into a Haskell compiler (GHC) for compilation to machine code or interpretation as byte code. One of the main advantages of this approach is that berp is able to use the rich functionality provided by the GHC runtime system with minimal implementation effort. Berp provides both a compiler and an interactive interpreter, and for the most part it can be used in the same way as CPython (the main Python implementation). Although berp is in the early stages of development, it is able to demonstrate some novel capabilities (compared to CPython), such as tail-call optimisation and call-with-current-continuation.

The syntactic analysis component of berp is provided by a separate Haskell library called language-python (→ 8.2.2), which can be used independently of berp to produce tools for processing Python source.

Berp underwent a flurry of development activity in the first part of 2010, but since then the pace slowed down as I worked on other projects. Those other projects are now maturing, and I plan to return to berp development soon. Berp is still missing support for some critical features, such as module imports, and I hope to remedy most of the major omissions by the end of the year.

Further reading

- http://hackage.haskell.org/package/berp
- http://github.com/bjpop/berp
- http://github.com/bjpop/berp/wiki

5.4.4 LQPL — A Quantum Programming Language Compiler and Emulator

Report by:	Brett G. Giles
Participants:	Dr. J.R.B. Cockett
Status:	v 0.8.4 experimental released

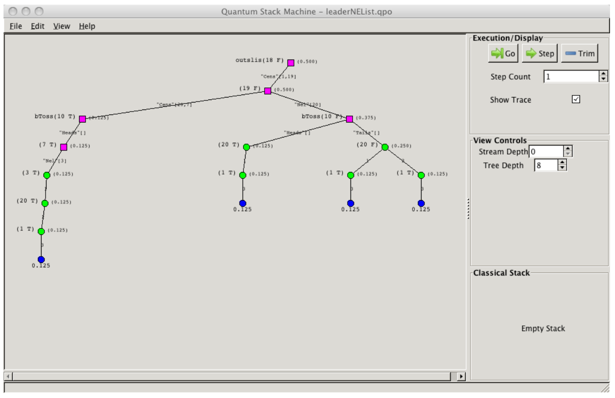
LQPL (Linear Quantum Programming Language) consists of a compiler for a functional quantum programming language and an associated assembler and emulator.

This programming language was inspired by Peter Selinger's paper "Toward a Quantum Programming Language". LQPL incorporates a simplified module / include system (more like C's include than Haskell's import), predefined unitary transforms, quantum control and classical control, algebraic data types, and operations on purely classical data.

Quantum programming allows us to provide a fair coin toss, as shown in the code example below.

```
qdata Coin       = {Heads | Tails}
toss ::( ; c:Coin) =
{  q = |0>;      Had q;
   measure q of
     |0> => {c = Heads}
     |1> => {c = Tails}
}
```

This allows programming of various probabilistic algorithms, such as leader election. The picture below is a screenshot of the emulator part way through leader election, showing a probabilistic list (`outslis`) with equal chances of being one of $[3,2]$ or $[3,1]$ and a coin toss (`bToss`) with equal chances of being Heads or Tails.



Quantum Stack Machine – leaderNEList.qpo

Step Count 1

Show Trace ✓

Stream Depth 0

Tree Depth 8

Classical Stack

Empty Stack

Work on version 0.9 has begun, with the primary goal of further de-coupling the emulator from the user interface. Currently, the user display, the emulator and the assembler are in a monolithic form. Once de-coupled, the intent is to allow the emulator to run independently of the display. This will allow a greater allocation of resources to the emulator, and allow the development of alternate display visualizations.

Further reading

http://pll.cpsc.ucalgary.ca/lqpl/index.html

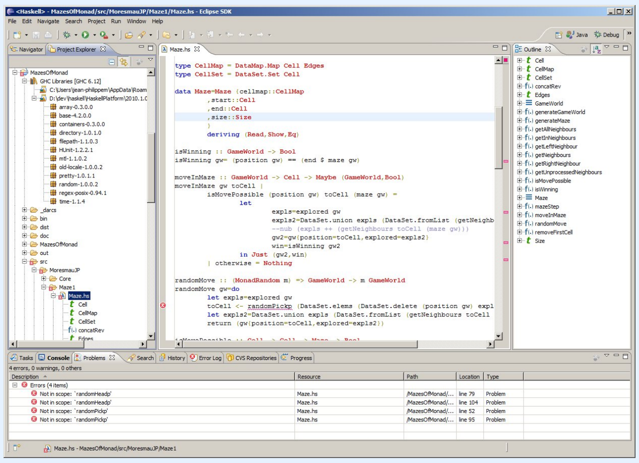
6 Development Tools

6.1 Environments

6.1.1 EclipseFP

Report by:	JP Moresmau
Participants:	Scott B. Michel, building on code from Thiago Arrais, Leif Frenzel, Thomas ten Cate, and others
Status:	stable, maintained

EclipseFP is a set of Eclipse plugins to allow working on Haskell code projects. It features Cabal integration (.cabal file editor, uses Cabal settings for compilation), and GHC integration. Compilation is done via the GHC API, syntax coloring uses the GHC Lexer. Other standard Eclipse features like code outline, folding, and quick fixes for common errors are also provided. EclipseFP also allows launching GHCi sessions on any module including extensive debugging facilities. It uses Scion to bridge between the Java code for Eclipse and the Haskell APIs. The source code is fully open source (Eclipse License) and anyone can contribute. Current version is 2.0.1, released in October 2010, and more versions with additional features are planned. Feedback on what is needed is welcome! The website has information on downloading binary releases and getting a copy of the source code. Support and bug tracking is handled through Sourceforge forums.



```
type CellMap = DataMap.Map Cell Edges
type CellSet = DataSet.Set Cell

data Maze=Maze {cellmap::CellMap
        ,start::Cell
        ,end::Cell
        ,size::Size
        }
        deriving (Read,Show,Eq)

isWinning :: GameWorld -> Bool
isWinning gw= (position gw) == (end $ maze gw)

moveInMaze :: GameWorld -> Cell -> Maybe (GameWorld,Bool)
moveInMaze gw toCell |
        isMovePossible (position gw) toCell (maze gw) =
                let
                        expls=explored gw
                        expls2=DataSet.union expls (DataSet.fromList (getNeighb
                        --nub (expls ++ (getNeighbours toCell (maze gw)))
                        gw2=gw{position=toCell,explored=expls2}
                        win=isWinning gw2
                in Just (gw2,win)
        | otherwise = Nothing

randomMove :: (MonadRandom m) => GameWorld -> m GameWorld
randomMove gw=do
        let expls=explored gw
        toCell <- randomPickp (DataSet.elems (DataSet.delete (position gw) expl
        let expls2=DataSet.union expls (DataSet.fromList (getNeighbours toCell
        return (gw{position=toCell,explored=expls2})
```

4 errors, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (4 items)				
Not in scope: `randomHeadp'	Maze.hs	/MazesOfMonad/...	line 79	Problem
Not in scope: `randomHeadp'	Maze.hs	/MazesOfMonad/...	line 104	Problem
Not in scope: `randomPickp'	Maze.hs	/MazesOfMonad/...	line 52	Problem
Not in scope: `randomPickp'	Maze.hs	/MazesOfMonad/...	line 95	Problem

Maze.hs - MazesOfMonad/src/MoresmauJP/Maze1

Further reading

http://eclipsefp.sourceforge.net/

6.1.2 ghc-mod — Happy Haskell Programming on Emacs

Report by:	Kazu Yamamoto
Status:	open source, actively developed

`ghc-mod` is an enhancement of Haskell mode on Emacs. It provides the following features:

Completion You can complete a name of keyword, module, class, function, types, language extensions, etc.

Code template You can insert a code template according to the position of the cursor. For instance, "module Foo where" is inserted in the beginning of a buffer.

Syntax check Code lines with error messages are automatically highlighted thanks to flymake. You can display the error message of the current line in another window. `hlint` can be used instead of GHC to check Haskell syntax.

Document browsing You can browse the module document of the current line either locally or on Hackage.

`ghc-mod` consists of code in Emacs Lisp and a sub-command in Haskell. The Emacs code executes the sub-command to obtain information about your Haskell environment. The sub-command makes use of GHC API for that purpose.

Further reading

http://www.mew.org/~kazu/proj/ghc-mod/en/

6.1.3 Leksah — Toward a Haskell IDE

Report by:	Jürgen Nicklisch-Franken

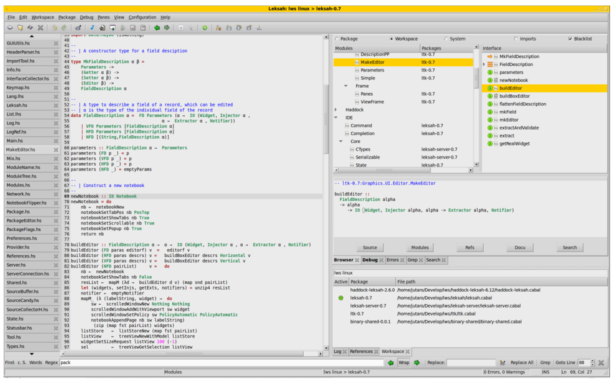
Leksah is a Haskell IDE written in Haskell, it uses Gtk+, and runs on Linux, Windows, and Mac OS X. Leksah is intended to be a practical tool to support the Haskell development process. Leksah is completely free.

Some features of Leksah:

- It uses the cabal package format and incorporates a cabal file editor.
- It offers Workspaces for complex projects with multiple packages with automatic build of dependencies.
- It contains a module browser that allows you to find type information about all the functions/symbols

available in the packages installed on your system.

- For most packages it shows as well haddock style comments, and gives direct navigation to sources.
- It integrates ghci debugging (including continuous recompilation) that allows you to type check and evaluate highlighted code snippets from within the editor itself. Includes a scratch buffer for testing ideas.
- It includes a helper for automatic addition of import statements.
- Offers a Haskell-customized editor with "source candy".
- Multi-window support for a multi head setting.
- Many standard features of IDEs like: Jump to errors, Auto Completion, Grep integration, . . .
- Configurable with session support, keymaps, and flexible appearance.



```

--
-- | A constructor type for a field desciption
--
type MkFieldDescription α β =
    Parameters ->
    (Getter α β) ->
    (Setter α β) ->
    (Editor β) ->
    FieldDescription α

--
-- | A type to describe a field of a record, which can be edited
-- | α is the type of the individual field of the record
data FieldDescription α =  FD Parameters (α →  IO (Widget, Injector α ,
                                    α →  Extractor α , Notifier))
    | VFD Parameters [FieldDescription α]
    | HFD Parameters [FieldDescription α]
    | NFD [(String,FieldDescription α)]

parameters :: FieldDescription α →  Parameters
parameters (FD p _) = p
parameters (VFD p _) = p
parameters (HFD p _) = p
parameters (NFD _) = emptyParams

--
-- | Construct a new notebook
--
newNotebook :: IO Notebook
newNotebook = do
    nb ←  notebookNew
    notebookSetTabPos nb PosTop
    notebookSetShowTabs nb True
    notebookSetScrollable nb True
    notebookSetPopup nb True
    return nb

buildEditor :: FieldDescription α →  α →  IO (Widget, Injector α , α →  Extractor α , Notifier)
buildEditor (FD paras editorf) v  =   editorf v
buildEditor (HFD paras descrs) v =   buildBoxEditor descrs Horizontal v
buildEditor (VFD paras descrs) v =   buildBoxEditor descrs Vertical v
buildEditor (NFD pairList)     v =   do
    nb ←  newNotebook
    notebookSetShowTabs nb False
    resList ←  mapM (λd →  buildEditor d v) (map snd pairList)
    let (widgets, setInjs, getExts, notifiers) = unzip4 resList
    notifier ←  emptyNotifier
    mapM_ (λ (labelString, widget) →  do
        sw ←  scrolledWindowNew Nothing Nothing
        scrolledWindowAddWithViewport sw widget
        scrolledWindowSetPolicy sw PolicyAutomatic PolicyAutomatic
        notebookAppendPage nb sw labelString)
         (zip (map fst pairList) widgets)
    listStore   ←  listStoreNew (map fst pairList)
    listView    ←  treeViewNewWithModel listStore
    widgetSetSizeRequest listView 100 (-1)
    sel         ←  treeViewGetSelection listView
```

Modules	Packages
DescriptionPP	ltk-0.7
MakeEditor	ltk-0.7
Parameters	ltk-0.7
Simple	ltk-0.7
Frame	
Panes	ltk-0.7
ViewFrame	ltk-0.7
Haddock	
IDE	
Command	leksah-0.7
Completion	leksah-0.7
Core	
CTypes	leksah-server-0.7
Serializable	leksah-server-0.7
State	leksah-0.7

Interface
MkFieldDescription
FieldDescription
parameters
newNotebook
buildEditor
buildBoxEditor
flattenFieldDescription
mkField
mkEditor
extractAndValidate
extract
getRealWidget

```
-- ltk-0.7:Graphics.UI.Editor.MakeEditor

buildEditor ::
  FieldDescription alpha
  -> alpha
    -> IO (Widget, Injector alpha, alpha -> Extractor alpha, Notifier)
```

lws linux

Active	Package	File path
	haddock-leksah-2.6.0	/home/jutaro/Develop/lws/haddock-leksah-6.12/haddock-leksah.cabal
●	leksah-0.7	/home/jutaro/Develop/lws/leksah/leksah.cabal
	leksah-server-0.7	/home/jutaro/Develop/lws/leksah-server/leksah-server.cabal
	ltk-0.7	/home/jutaro/Develop/lws/ltk/ltk.cabal
	binary-shared-0.0.1	/home/jutaro/Develop/lws/binary-shared/binary-shared.cabal

Find: pack

Future plans

- Enhance usability and fix open bugs for the 1.0 release.
- Concept and implementation of an extension mechanism.
- Better integration of Yi as editor component.

The project needs more users and developers!

Further reading

http://leksah.org/

6.1.4 HEAT: The Haskell Educational Advancement Tool

Report by:	Olaf Chitil
Status:	active

Heat is an interactive development environment (IDE) for learning and teaching Haskell. Heat was designed for novice students learning the functional programming language Haskell. Heat provides a small number of supporting features and is easy to use. Heat is portable, small and works on top of the Haskell interpreter Hugs.

Heat provides the following features:

- Editor for a single module with syntax-highlighting and matching brackets.
- Shows the status of compilation: non-compiled; compiled with or without error.
- Interpreter console that highlights the prompt and error messages.
- If compilation yields an error, then the source line is highlighted and additional error explanations are provided.
- Shows a program summary in a tree structure, giving definitions of types and types of functions.
- Automatic checking of all (Boolean) properties of a program; results shown in summary.

A complete re-write of the current version 3.1 is planned to improve the internal structure and make Heat work with GHC.

Further reading

http://www.cs.kent.ac.uk/projects/heat/

6.1.5 HaRe — The Haskell Refactorer

Report by:	Simon Thompson
Participants:	Huiqing Li, Chris Brown, Claus Reinke

Refactorings are source-to-source program transformations which change program structure and organization, but not program functionality. Documented in catalogs and supported by tools, refactoring provides the means to adapt and improve the design of existing code, and has thus enabled the trend towards modern agile software development processes.

Our project, *Refactoring Functional Programs*, has as its major goal to build a tool to support refactorings in Haskell. The HaRe tool is now in its sixth major release. HaRe supports full Haskell 98, and is integrated with (X)Emacs and Vim. All the refactorings that HaRe supports, including renaming, scope change, generalization and a number of others, are *module-aware*, so that a change will be reflected in all the modules in a project, rather than just in the module where the change is initiated. The system also contains a set of data-oriented refactorings which together transform a concrete `data` type and associated uses of pattern matching into an abstract type and calls to assorted functions. The latest snapshots support the hierarchical modules extension, but only small parts of the hierarchical libraries, unfortunately.

In order to allow users to extend HaRe themselves, HaRe includes an API for users to define their own program transformations, together with Haddock documentation. Please let us know if you are using the API.

Snapshots of HaRe are available from our webpage, as are related presentations and publications from the group (including LDTA'05, TFP'05, SCAM'06, PEPM'08, PEPM'10, TFP'10, Huiqing's PhD thesis and Chris's PhD thesis). The final report for the project appears there, too.

Recent developments

- HaRe 0.6, which is compatible with GHC-6.12.1, has been released; HaRe 0.6 is available on Hackage, and also downloadable from our project webpage.
- HaRe 0.6 comes with a number of new refactorings, including adding and removing fields and constructors to data-type definitions, folding and unfolding against as-patterns, merging and splitting function definitions, converting between let and where constructs, introducing pattern matching and generative folding.
- Support for automatic detection and semi-automatic elimination of duplicated code in Haskell programs is also available from HaRe 0.6.

Further reading

http://www.cs.kent.ac.uk/projects/refactor-fp/

6.2 Documentation

6.2.1 Haddock

Report by:	David Waern
Status:	experimental, maintained

Haddock is a widely used documentation-generation tool for Haskell library code. Haddock generates documentation by parsing and typechecking Haskell source code directly and including documentation supplied by the programmer in the form of specially-formatted comments in the source code itself. Haddock has direct support in Cabal (→ 6.7.1), and is used to generate the documentation for the hierarchical libraries that come with GHC, Hugs, and nhc98 (http://www.haskell.org/ghc/docs/latest/html/libraries) as well as the documentation on Hackage.

The latest release is version 2.8.1, released September 3 2010.

Recent changes:

- HTML backend completely rewritten to generate semantically rich XHTML using the xhtml package.
- New default CSS based on the color scheme chosen for the new Haskell wiki, with a pull-out tab for the synopsis.
- Theme engine based on CSS files. Themes can be switched from the header menu.
- Markup support for executable examples/unit-tests.
- Addition of a LaTeX backend.
- Additions and changes to the Haddock API.
- Various smaller new features and bug fixes.

Future plans

- Although Haddock understands many GHC language extensions, we would like it to understand all of them. Currently there are some constructs you cannot comment, like GADTs and associated type synonyms.
- Error messages is an area with room for improvement. We would like Haddock to include accurate line numbers in markup syntax errors.
- On the HTML rendering side we want to make more use of Javascript in order to make the viewing experience better. The frames-mode could be improved this way, for example.
- Finally, the long term plan is to split Haddock into one program that creates data from sources, and separate backend programs that use that data via the Haddock API. This will scale better, not requiring adding new backends to Haddock for every tool that needs its own format.

Further reading

- Haddock's homepage: http://www.haskell.org/haddock/
- Haddock's developer Wiki and Trac: http://trac.haskell.org/haddock
- Haddock's mailing list: haddock@projects.haskell.org

6.2.2 Hoogle

Report by:	Neil Mitchell
Status:	stable

Hoogle is an online Haskell API search engine. It searches the functions in the various libraries, both by name and by type signature. When searching by name, the search just finds functions which contain that name as a substring. However, when searching by types it attempts to find any functions that might be appropriate, including argument reordering and missing arguments. The tool is written in Haskell, and the source code is available online. Hoogle is available as a web interface, a command line tool, and a lambdabot plugin.

Hoogle development has recently restarted, and work is proceeding quickly. The darcs version of Hoogle can now search all of Hackage (→ 6.7.1), and should be released in a few months.

Further reading

http://haskell.org/hoogle

6.2.3 lhs2TEX

Report by:	Andres Löh
Status:	stable, maintained

This tool by Ralf Hinze and Andres Löh is a preprocessor that transforms literate Haskell code into LATEX documents. The output is highly customizable by means of formatting directives that are interpreted by lhs2TEX. Other directives allow the selective inclusion of program fragments, so that multiple versions of a program and/or document can be produced from a common source. The input is parsed using a liberal parser that can interpret many languages with a Haskell-like syntax, and does not restrict the user to Haskell 98.

The program is stable and can take on large documents.

Since version 1.14, lhs2TEX has an experimental mode for typesetting Agda code.

The current version is 1.16, it should address the issues previous versions had with Unicode and should work properly using ghc-6.12. For the future, I plan a more substantial rewrite of lhs2TEX to clean up the internals and make the functionality of lhs2TEX available as a library.

Further reading

http://www.andres-loeh.de/lhs2tex

6.3 Testing and Analysis

6.3.1 HTF: A Test Framework for Haskell

Report by:	Stefan Wehr
Status:	beta, active development

The Haskell Test Framework (*HTF* for short) lets you define unit tests, QuickCheck properties, and black box tests in an easy and convenient way. The HTF uses a custom preprocessor that collects test definitions automatically. Furthermore, the preprocessor allows the HTF to report failing test cases with exact file name and line number information.

Initially created in 2005, HTF was not actively developed for almost five years. Development resumed in 2010, adding many improvements to the code base.

Further reading

- http://hackage.haskell.org/package/HTF
- Tutorial: http://www.factisresearch.com/2010/03/htf/

6.3.2 SourceGraph

Report by:	Ivan Lazar Miljenovic
Status:	version 0.6.1.1

SourceGraph is a utility program aimed at helping Haskell programmers visualize their code and perform simple graph-based analysis (representing entities as nodes in the graphs and function calls as directed edges), which started off as an example of how to use the Graphalyze library (→ 8.4.3), which is designed as a general-purpose graph-theoretic analysis library. These two pieces of software were originally developed as the focus of my mathematical honors thesis, "Graph-Theoretic Analysis of the Relationships Within Discrete Data".

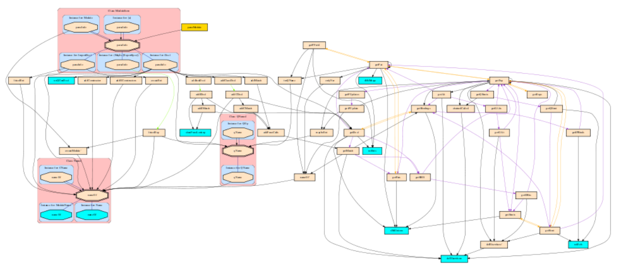
Whilst fully usable, SourceGraph is currently limited in terms of input and output. It analyses all `.hs` and `.lhs` files recursively found in the provided directory, parsing most aspects of Haskell code (cannot parse Haskell code using CPP, HaRP, TH, FFI and XML-based Haskell code; difficulty parsing Data Family instances, unknown modules and record puns and wildcards). The results of the analysis are created in an Html file in a "SourceGraph" subdirectory of the project's root directory.

Various refinements have been implemented since the last release, including:

- "Implicitly exported" entities (e.g., class method instance definitions from external classes) are now supported; support for these is not perfect and may include more entities than it should.
- Addition of depth analysis (based upon how many function calls are needed from an exported entity).
- Better visualizations, including edge categorizations; the generated Dot code is also saved if users wish to tweak these.

Current analysis algorithms utilized include: alternative module groupings, whether a module should be split up, root analysis, depth analysis, clique and cycle detection, as well as finding functions which can safely be compressed down to a single function. Please note however that SourceGraph is *not* a refactoring utility, and that its analyses should be taken with a grain of salt: for example, it might recommend that you split up a module, because there are several distinct groupings of functions, when that module contains common utility functions that are placed together to form a library module (e.g., the Prelude).

Sample SourceGraph analysis reports can be found at http://code.haskell.org/~ivanm/Sample_SourceGraph/SampleReports.html. A tool paper on SourceGraph was presented at the ACM SIGPLAN 2010 Workshop on Partial Evaluation and Program Manipulation.



Further reading

- http://hackage.haskell.org/package/SourceGraph
- http://ivanmiljenovic.files.wordpress.com/2008/11/honoursthesis.pdf
- http://ivanmiljenovic.files.wordpress.com/2010/03/sourcegraph_pepm10_reprint.pdf

6.3.3 HLint

Report by:	Neil Mitchell
Status:	stable

HLint is a tool that reads Haskell code and suggests changes to make it simpler. For example, if you call `maybe foo id` it will suggest using `fromMaybe foo` instead. HLint is compatible with almost all Haskell extensions, and can be easily extended with additional hints.

There have been numerous feature improvements since the last HCAR, including an API, and better support for qualified module names. HLint is now used within hpaste.org.

Further reading

http://community.haskell.org/~ndm/hlint/

6.3.4 A Haskell Source File Scanning Tool

Report by:	Christian Maeder
Status:	maintained

The Haskell source file scanning tool `scan` is supposed to be a complement for `hlint` (→6.3.3). Whereas the latter makes suggestions to improve your expressions, `scan` makes suggestions about your source file format regarding white spaces, layout and comments, as usually described by style guides.

An extra feature of `scan` is to write back an adjusted source file that is untabified, has no trailing white space, and ends with a single final newline. The motivating purpose was to put blanks around infix operators and after commas. But also multiple blanks and comments may be adjusted controlled by command line options. A new option is to change consecutive line comments to a single block comment. More options may follow.

Although in some cases layout may be destroyed and modifications should be reverted or made manually, this tool is supposed to keep Haskell sources tidy.

Further reading

http://projects.haskell.org/style-scanner/

6.4 Boilerplate Removal

6.4.1 A Generic Deriving Mechanism for Haskell

Report by:	José Pedro Magalhães
Participants:	Atze Dijkstra, Johan Jeuring, Andres Löh
Status:	actively developed

Haskell's deriving mechanism supports the automatic generation of instances for a number of functions. The Haskell 98 Report only specifies how to generate instances for the Eq, Ord, Enum, Bounded, Show, and Read classes. The description of how to generate instances is largely informal. As a consequence, the portability of instances across different compilers is not guaranteed. Additionally, the generation of instances imposes restrictions on the shape of datatypes, depending on the particular class to derive.

We have developed a new approach to Haskell's deriving mechanism, which allows users to specify how to derive arbitrary class instances using standard datatype-generic programming techniques. Generic functions, including the methods from six standard Haskell 98 derivable classes, can be specified entirely within Haskell 98 plus multi-parameter type classes, making them lightweight and portable. We can also express *Functor*, *Typeable*, and many other derivable classes with our technique.

We have implemented our deriving mechanism together with many new derivable classes in the Utrecht Haskell Compiler (→3.5). Currently we are working on implementing it in GHC as well, replacing the existing (but rarely used) generic classes. The underlying library for generic data representation, `generic-deriving`, is available on Hackage.

Further reading

http://dreixel.net/research/pdf/gdmh.pdf

6.4.2 Derive

Report by:	Neil Mitchell
Status:	v2.3.0

The Derive tool is used to generate formulaic instances for data types. For example given a data type, the Derive tool can generate 34 instances, including the standard ones (Eq, Ord, Enum etc.) and others such as Binary and Functor. Derive can be used with SYB, Template Haskell or as a standalone preprocessor. This tool serves a similar role to DrIFT, but with additional features.

Recently Derive has had many derivations added, including new Uniplate (→ 8.5.2) instances. The mechanism to derive instances by example has been rewritten, and the revised mechanism is described in the associated Approaches and Applications of Inductive Programming 2009 paper.

Further reading

http://community.haskell.org/~ndm/derive/

6.4.3 Agata

Report by:	Jonas Duregård
Participants:	Koen Claessen
Status:	experimental, active

The Agata library (Agata Generates Algebraic Types Automatically) is an outcome of my master's thesis work at Chalmers University of Technology. The library uses Template Haskell to derive instances of the QuickCheck `Arbitrary` class for (almost) any Haskell data type.

The generators differ from regular QuickCheck generators in that they maintain scalability even for types analogous to nested collection data structures (e.g., `[[[[a]]]]`, where the standard QuickCheck generator tends to generate values that contain millions of `a`'s). Generators also guarantee that independent components of the same type have the same expected size, e.g., in `(a,[a])` the single `a` will have the same expected size as any `a` in the list.

Although a few additional features are to be implemented in the near future, efforts will be focused on documentation and improving performance. When the library is stable and well documented, the possibility of integrating it into the QuickCheck package may be explored.

Further reading

- http://hackage.haskell.org/package/Agata
- Agata — Random generation of test data (Master's thesis), http://gupea.ub.gu.se/bitstream/2077/22087/1/gupea_2077_22087_1.pdf

6.5 Code Management

6.5.1 Darcs

Report by:	Eric Kow
Participants:	darcs-users list
Status:	active development

Darcs is a distributed revision control system written in Haskell. In Darcs, every copy of your source code is a full repository, which allows for full operation in a disconnected environment, and also allows anyone with read access to a Darcs repository to easily create their own branch and modify it with the full power of Darcs' revision control. Darcs is based on an underlying theory of patches, which allows for safe reordering and merging of patches even in complex scenarios. For all its power, Darcs remains a very easy to use tool for every day use because it follows the principle of keeping simple things simple.

Our most recent major release, Darcs 2.5, was in November 2010. It provides faster repository-local operations, faster record with long patch histories, among other bug fixes and features.

Since our last report, we have had two successful Google Summer of Code projects in the 2010 programme. We hope that the work from these projects will make it into our next major release Darcs 2.8, scheduled for March 2011:

Adolfo Builes improved the reliability of the Darcs cache system, making Darcs performance more predictable. He used his work as a basis for a high-level documentation effort (http://wiki.darcs.net/Internals/CacheSystem), explaining the technical details behind Darcs without implementation detail.

Alexey Levan optimised the darcs get operation with an `optimize --http` command. In a recent test, we found that this dramatically reduced the time to fetch Darcs' own repository:

	before	after
`get`	40 min	3 min
`get --lazy`	2 min	0.2 min (11s)

Meanwhile, we still have a lot progress to make and are always open to contributions. Haskell hackers, we need your help!

Darcs is free software licensed under the GNU GPL. Darcs is a proud member of the Software Freedom Conservancy, a US tax-exempt 501(c)(3) organization. We accept donations at http://darcs.net/donations.html.

Further reading

http://darcs.net

6.5.2 ipatch

Report by:	Joachim Breitner
Status:	working

ipatch brings some of Darcs' specialities, most notably the hunk selection and editing interface, to those who work with plain patch files outside any version control system. Currently, it allows you to interactively and selectively apply a patch or to split a patch into several patch files.

ipatch has not seen a lot of use yet and certainly has rough edges. It can nevertheless be useful already. It can be installed from hackage, and patches are, as always, welcome.

Further reading

- http://hackage.haskell.org/package/ipatch
- https://www.joachim-breitner.de/blog/archives/425-ipatch,-the-interactive-patch-editor.html

6.5.3 DarcsWatch

Report by:	Joachim Breitner
Status:	working

DarcsWatch is a tool to track the state of Darcs (→ 6.5.1) patches that have been submitted to some project, usually by using the `darcs send` command. It allows both submitters and project maintainers to get an overview of patches that have been submitted but not yet applied.

DarcsWatch continues to be used by the xmonad project (→ 9.8.1), the Darcs project itself, and a few developers. At the time of writing, it was tracking 42 repositories and 3557 patches submitted by 189 users.

Further reading

- http://darcswatch.nomeata.de/
- http://darcs.nomeata.de/darcswatch/documentation.html

6.5.4 DPM — Darcs Patch Manager

Report by:	Stefan Wehr
Participants:	David Leuschner
Status:	beta, active development

The Darcs Patch Manager (*DPM* for short) is a tool that simplifies working with the revision control system darcs (http://darcs.net). It is most effective when used in an environment where developers do not push their patches directly to the main repository but where patches undergo a reviewing process before they are actually applied.

The current feature set of DPM is quite stable. In our company (→ 10.4), we actively use DPM to keep track of all patches sent to various projects. At the Haskell hackathon 2010 in Zürich, we started working on support for tracking conflicts between patches. We did not yet finish this work, but hope to provide a new DPM release with support for conflicts in May 2010.

There is some overlap between DPM and darcswatch (→ 6.5.3). The main difference between darcswatch and DPM is that the former mainly targets developers whereas the latter helps reviewers doing their work.

Further reading

- http://hackage.haskell.org/package/DPM
- Tutorial: http://www.factisresearch.com/2010/03/dpm/

6.6 Interfacing to other Languages

6.6.1 HSFFIG

Report by:	Dmitry Golubovsky
Status:	release

Haskell FFI Binding Modules Generator (HSFFIG) is a tool which parses C include files (`.h`) and generates Haskell Foreign Functions Interface import declarations for all functions, suitable `#define`'d constants, enumerations, and structures/unions (to access their members). It is assumed that the GNU C Compiler and Preprocessor are used. Auto-generated Haskell modules may be imported into applications to access foreign libraries' functions and variables in type-safe manner.

In the current version 1.1.3, speed of processing `#define`'d constants is considerably improved by using HSFFIG's own C language syntax parser to determine suitability of constants for FFI import. Previous versions of HSFFIG invoked an external C compiler for this purpose.

Further reading

- The HSFFIG package on Hackage
  http://hackage.haskell.org/package/HSFFIG
- The HSFFIG Tutorial
  http://www.haskell.org/haskellwiki/HSFFIG/Tutorial
- The FFI Imports Packaging Utility
  http://www.haskell.org/haskellwiki/FFI_imports_packaging_utility

6.6.2 Hubris

Report by:	Mark Wotton
Participants:	James Britt, Larry Diehl, Josh Price, Tatsuhiro Ujihisa, Andrew Grimm
Status:	beta

Hubris is an in-process bridge between Ruby and Haskell, allowing Ruby programs to use Haskell code without writing boilerplate.

It is now easier to install, and some 64 bit bugs have been fixed.

To get it on Linux:

```
cabal install hubris
gem install hubris
```

Mac OS X is a bit harder because support for dynamic libraries has not been merged into the GHC mainline yet, but it is in the pipe. Further plans:

- work with new versions of Ruby without reinstallation of Hubris Haskell-side support code
- support for passing RTS flags to the Haskell process
- translation instance injection (i.e., express equivalents for complex Haskell datatypes in Ruby and vice versa)
- multiple argument support
- some way of storing non-translatable instances on the ruby side — ideally, you should be able to have a Ruby list of Haskell functions, and apply each of them in turn. Currently only translateable data types are marshalled.

Further reading

- http://github.com/mwotton/Hubris-Haskell
- http://github.com/mwotton/Hubris
- http://www.engineyard.com/blog/2010/a-hint-of-hubris/
- http://www.jamesbritt.com/2010/3/13/a-purely-functional-tale-of-a-bridge-compose-of-hubris

6.7 Deployment

6.7.1 Cabal and Hackage

Report by: Duncan Coutts

Background

Cabal is the Common Architecture for Building Applications and Libraries. It defines a common interface for defining and building Haskell packages. It is implemented as a Haskell library and associated tools which allow developers to easily build and distribute packages.

Hackage is a distribution point for Cabal packages. It is an online database of Cabal packages which can be queried via the website and client-side software such as cabal-install. Hackage enables end-users to download and install Cabal packages.

cabal-install is the command line interface for the Cabal and Hackage system. It provides a command line program `cabal` which has sub-commands for installing and managing Haskell packages.

Recent progress

There will shortly be a new release of Cabal-1.10 and cabal-install-0.10. They will be available from hackage and will be included in the next major release of the Haskell Platform (→ 3.1).

The major new feature for Cabal-1.10 is "cabal test". This is a feature to allow packages to define test suites. There is an interface to allow build agents such as `cabal` or a hackage buildbot to run the testsuites and collect the results. This feature is the result of a Google Summer of Code project by Thomas Tuegel (→ 7.2).

Matt Gruen successfully completed a GSoC project this summer which was to improve the new `hackage-server` implementation (→ 7.4).

Looking forward

The new `hackage-server` is now nearly ready for live deployment, but some work remains to be done on the transition from the old server to the new. Volunteering to help with this process would be a great service to the community.

There are many improvements we want to make to Cabal, cabal-install, and Hackage. I am pleased to report that we have had an increase in new contributors in the last few months, however our limiting factor remains the amount of volunteer development time and also maintainer code-review time. We would like to encourage people considering contributing to join the `cabal-devel` mailing list so that we can increase development discussion and improve collaboration. The bug tracker is well maintained and it should be relatively clear to new contributors what is in need of attention and which tasks are considered relatively easy.

Further reading

- Cabal homepage: http://www.haskell.org/cabal
- Hackage package collection: http://hackage.haskell.org/
- Bug tracker: http://hackage.haskell.org/trac/hackage/

6.7.2 Capri

Report by: Dmitry Golubovsky
Status: experimental

Capri (abbreviation of **CA**bal **PRI**vate) is a wrapper program on top of cabal-install to operate it in project-private mode. In this mode, there is no global or user package databases; only one package database is defined, private to the project, located under the root directory of a project.

Capri invokes cabal-install and ghc-pkg in the way that only a project's private package database is visible to them. Starting with a minimally required set of packages, all necessary dependencies will be installed per project, not affecting user or global databases. This helps maintain a clean build environment without the risk of accidental installation of conflicting versions of the same package which sometimes happens with the global packages database.

Capri is mainly intended to build executable programs. It depends on certain features of GHC, and is not usable with other Haskell compilers.

Further reading

- The Capri package on Hackage
  http://hackage.haskell.org/package/capri
- The Capri Tutorial
  http://www.haskell.org/haskellwiki/Capri

6.7.3 Shaker

Report by:	Anthonin Bonnefoy
Status:	active development

Shaker is an interactive build tool which allows to compile and execute tests on a Haskell project and provides several features like:

- Continuous mode: In continuous mode, an action (compile or test) is triggered by source changes.
- Automatic test discovery: Shaker discovers and executes tests via the GHC API. All exported QuickCheck properties and HUnit test cases can be executed by Shaker.
- Selectable test execution: One or several tests can be selected for execution using regular expressions.
- `test-framework` integration for test execution
- Easy configuration: Shaker reuses cabal configuration so there is no need for extra configuration if your project is already cabalized.

Shaker can be used to type check your code as you edit it; With the `~compile` command, a compilation will be executed as soon as a source change is detected. You can also execute a specific test on source change with `~test aTestName`.

Future plans

Shaker is incompatible with several projects due to some special cases not managed, and current development aims to make it compatible with more projects. After this, the next feature will be the possibility to execute only previously failing tests.

Further reading

- http://hackage.haskell.org/package/shaker
- http://github.com/bonnefoa/Shaker

7 Google Summer of Code 2010

7.1 Immix Garbage Collector on GHC

Report by:	Marco Silva
Status:	unconcluded

During the summer of 2010, Marco Silva worked on the implementation of the Immix algorithm in GHC. Immix is a relatively new technique for garbage collection, which has been shown to be better than other alternatives, including the ones used in GHC. The work was done as a project in the Google Summer of Code.

The code is functional and does not contain known bugs. It gets better results than the default GC in the nofib suite. On the other hand, it gets worse results than the default GC for the nofib/gc suite. This scenario may change if more tuning is done in the details of the implementation. Given that GHC allows the user to choose between garbage collection alternatives at runtime, it is easy to test and compare the different techniques.

Immix was implemented using the experimental code from mark-sweep as a base. Currently, it overrides mark-sweep so that it is not that easy to compare immix with mark-sweep. The plan is to split them apart in the future.

On the GHC Commentary there is a page about the current state, with a to do list. There are some fundamental parts that are not yet implemented and that may improve performance, such as the allocation in lines in minor GCs and the removal of partial lists, which are not necessary in Immix.

Further reading

http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/GC/Immix

7.2 Improvements to Cabal's Test Support

Report by:	Thomas Tuegel
Participants:	Johan Tibell (Mentor)
Status:	active development

As part of the Google Summer of Code 2010, Cabal's test support was improved to allow automated testing of packages. The intent is to provide the technical enhancements necessary for wide adoption of automatic testing in Haskell software, improving the software's general quality. The results of the Summer of Code project were presented at the 2010 Haskell Implementors Workshop, but work is ongoing.

A basic test interface allowing package authors to specify standalone test executables in their package description files will be available in Cabal 1.10 (→ 6.7.1). Cabal can run these tests from the command line and report the aggregate results of the test suite in human- and machine-readable format. Work is in progress to support a standard interface for modules containing multiple test cases; this will make it possible for Cabal to report on the results of individual cases within a test suite. Future work on Hackage will make test reports for uploaded packages available automatically.

Further reading

- http://cabaltest.blogspot.com
- http://haskell.org/haskellwiki/HaskellImplementorsWorkshop/2010

7.3 A High Performance HTML Generation Library

Report by:	Jasper Van der Jeugt
Status:	stable

See BlazeHtml (→ 8.8.4).

7.4 Hackage 2.0

Report by:	Matthew Gruen
Participants:	Duncan Coutts
Status:	in development

Hackage 2.0 is a rewrite of the original Hackage (→ 6.7.1) infrastructure intended to provide additional features and better handle Haskell's sustained growth. It was developed to a near-deployable state as part of the 2010 Google Summer of Code program. Enhancing Hackage's role as a package repository, it adds metrics for packages, means of communication between end-users and maintainers, and tools to aid quality assurance.

Currently, Hackage runs an Apache instance to store packages. It is very stable, but also difficult to extend. Plain text files are used to store information, so some features which require plenty of in-memory data manipulation are costly. The new codebase, called hackage-server (in contrast to the current hackage-scripts), uses the Happstack web framework (→ 5.2.6) for just about everything. It employs happstack-state to store native Haskell datatypes in-memory, with a separate file store for the package tarballs themselves.

Features

The primary design goal of hackage-server is to provide a modular, extensible infrastructure for any conceivable feature that might be added to Hackage. It has full feature parity with hackage-scripts and more, with a RESTful backend supporting multiple content formats.

Reverse dependencies, editable tags, and download counts have all been implemented to help locate useful libraries out of thousands. Deprecation, user-submitted build reports, and a user groups system are intended to make maintainance easier. There is also the ability to post packages on a beta-testing index before publishing it on Hackage proper.

Roadmap

The eventual goal is to have the hackage-server codebase serving packages at http://hackage.haskell.org. It is much closer to this now than half a year ago! Further work involves improving performance in both time and memory.

The first major deployment will be a simple mirror for the main Hackage on the sparky server (at sparky.haskell.org, port 8080) which Cabal can set as a remote repository. Afterwards, the mirror will be open for editing by anyone with an account on the main Hackage. The full switchover will occur as soon as we are confident about the stability.

Further reading

- Wiki documentation: http://hackage.haskell.org/trac/hackage/wiki/HackageDB/2.0
- Code: `darcs get` http://code.haskell.org/hackage-server

7.5 Improving Darcs' Network Performance

See Darcs (→ 6.5.1).

8 Libraries

8.1 Processing Haskell

8.1.1 The Neon Library

Report by:	Jurriaan Hage

As part of his master thesis work, Peter van Keeken implemented a library to data mine logged Helium (→ 3.4) programs to investigate aspects of how students program Haskell, how they learn to program, and how good Helium is in generating understandable feedback and hints. The software can be downloaded from http://www.cs.uu.nl/wiki/bin/view/Hage/Neon, which also gives some examples of output generated by the system. The downloads only contain a small sample of loggings, but it will allow programmers to play with it. This work has been continued by Mathijs Swint, but the results of his work have told us that although we do have a lot of data, we need quite a bit more in order to get significant results from Neon.

8.1.2 mueval

Report by:	Gwern Branwen
Participants:	Andrea Vezzosi, Daniel Gorin, Spencer Janssen, Adam Vogt
Status:	active development

Mueval is a code evaluator for Haskell; it employs the GHC API as provided by the Hint library (http://haskell.org/communities/11-2008/html/report.html#hint). It uses a variety of techniques to evaluate arbitrary Haskell expressions safely & securely. Since it was begun in June 2008, tremendous progress has been made; it is currently used in Lambdabot live in #haskell). Mueval can also be called from the command-line.

Mueval features:

- A comprehensive test-suite of expressions which should and should not work
- Defeats all known attacks
- Optional resource limits and module imports
- The ability to load in definitions from a specified file
- Parses Haskell expressions with haskell-src-exts and tests against black- and white-lists
- A process-level watchdog, to work around past and future GHC issues with thread-level watchdogs
- Cabalized

Since the November 2009 HCAR report, the internals have been cleaned up further, a number of minor bugs squashed, tests added, and mueval updated to avoid bitrot.

We are currently working on the following:

- Refactoring modules to render Mueval more useful as a library
- Removing the POSIX-only requirement
- Merging in Chris Done's mueval-interactive fork, which powers http://tryhaskell.org/

Further reading

The source repository is available: `darcs get` http://code.haskell.org/mubot/

8.2 Parsing and Transforming

8.2.1 The grammar-combinators Parser Library

Report by:	Dominique Devriese
Status:	partly functional

The grammar-combinators library is an experimental next-generation parser library written in Haskell (LGPL license). The library features much of the power of a parser generator like Happy or ANTLR, but with the library approach and most of the benefits of a parser combinator library.

The project's initial release was in September 2010. A paper about the main idea is being published at the PADL'11 conference and an accompanying technical report with more implementation details is available online. The library is published on Hackage under the name grammar-combinators.

We believe this library is an ideal place for innovations in practical functional parsing libraries. The library adds substantial fundamental power to traditional parser combinator libraries and opens up the path to implementing many parsing techniques that were previously impossible. We believe people interested in parsing techniques will find the library ideal for implementing their ideas and we encourage all contributions.

However, the library still needs a lot of love before it is suited for mainstream use. Performance is not ideal and many real-world features are missing. People interested to work on these topics are very welcome to contact us!

Further reading

http://projects.haskell.org/grammar-combinators/

8.2.2 language-python

Report by:	Bernie Pope
Status:	stable

Language-python is a Haskell library for lexical analysis, parsing, and pretty printing Python code. It supports versions 2.x and 3.x of Python. The parser is implemented using the happy parser generator, and the alex lexer generator. It supports source accurate span information and optional parsing of comments. A separate package called language-python-colour is available on Hackage which demonstrates the use of the library to render Python source in coloured XHTML. the library is also used for the syntactic analysis component of the berp Python compiler (→ 5.4.3).

Further reading

- http://hackage.haskell.org/package/language-python
- http://github.com/bjpop/language-python

8.2.3 Loker

Report by:	Roman Cheplyaka
Participants:	Andrey Torba, Evgeniy Berkovich, and others
Status:	in active development

Loker is a collection of programs to deal with UNIX Shell scripts. It will include a parser, a static analysis tool, and a compiler. The distinctive feature of the project is strong compliance to the POSIX standard. All the parts are written in Haskell.

At the moment the parser is almost complete and some work is being done on the analysis tool and the compiler.

The plan is to make the first release soon.

Further reading

http://github.com/feuerbach/loker

8.2.4 ChristmasTree

Report by:	Marcos Viera
Participants:	Doaitse Swierstra, Eelco Lempsink
Status:	experimental

See: http://haskell.org/communities/05-2009/html/report.html#sect5.5.7.

8.2.5 First Class Syntax Macros

Report by:	Marcos Viera
Participants:	Doaitse Swierstra, Atze Dijkstra, Arthur Baars
Status:	experimental

See: http://haskell.org/communities/05-2010/html/report.html#sect5.4.2.

8.2.6 Utrecht Parser Combinator Library: uu-parsinglib

Report by:	Doaitse Swierstra
Status:	actively developed

The `uu-parsinglib` library was extended with a unified approach to writing parsers which recognize several elements in arbitrary order. E.g., the parser:

$$\begin{array}{lll} (,,,)\ \text{`}pMerge\text{`} & (& pSome\ pa \\ & <||> & pMany\ pb \\ & <||> & pOne\ pc \\ & <||> & pNatural\ \text{`}pOpt\text{`}\ 5) \end{array}$$

when provided with the input `"45bab"` returns the value `(["a"],["b","b"],"c",45)` and the error message

```
-- > Result: (["a"],["b","b"],"c",45)
-- > Correcting steps:
-- >    Inserted 'c' at position (0,5)
        expecting one of ['b', 'a', 'c']
```

A tutorial has appeared in the LNCS lecture notes (S. Doaitse Swierstra, Combinator Parsers: A Short Tutorial, Language Engineering and Rigorous Software Development 2009, LNCS 5520). The text is also available as a technical report at http://www.cs.uu.nl/research/techreps/UU-CS-2008-044.html.

Furthermore the library was provided with many more examples in the `Examples` module, and lots of Haddock documentation was added.

Features

- Much simpler internals than the old library (http://haskell.org/communities/05-2009/html/report.html#sect5.5.8).
- Combinators for easily describing parsers which produce their results online, do not hang on to the input and provide excellent error messages.
- Parsers “correct” the input such that parsing can proceed when an erroneous input is encountered.
- The library provides both an applicative interface and a monadic interface.
- No need for *try*-like constructs which makes writing `Parsec` based parsers tricky.

- Scanners can be switched dynamically, so several different languages can occur intertwined in a single input file.

Future plans

The next version will contain a check for grammars being not left-recursive, thus taking away the only remaining source of surprises when using parser combinator libraries. This makes the library great for teaching environments too. Future versions of the library, using even more abstract interpretation, will make use of computed look-ahead information to speed up the parsing process further. Gradually software from Utrecht will be moving to use the new library `uu-parsinglib`.

Contact

If you are interested in using the current version of the library in order to provide feedback on the provided interface, contact ⟨doaitse@swierstra.net⟩. There is a low volume, moderated mailing list at ⟨parsing@cs.uu.nl⟩. More information can be found at http://www.cs.uu.nl/wiki/bin/view/HUT/ParserCombinators.

8.2.7 Regular Expression Matching with Partial Derivatives

Report by:	Martin Sulzmann
Participants:	Kenny Zhuo Ming Lu
Status:	stable

Regular expression matching is a classical and well-studied problem. Prior work applies DFA and Thompson NFA methods for the construction of the matching automata. We propose the novel use of derivatives and partial derivatives for regular expression matching. We show how to obtain algorithms for various matching policies such as POSIX and greedy left-to-right. Our benchmarking results show that the run-time performance is promising and that our approach can be applied in practice.

Further reading

- http://hackage.haskell.org/package/regex-pderiv
- http://sulzmann.blogspot.com/2010/04/regular-expression-matching-using.html

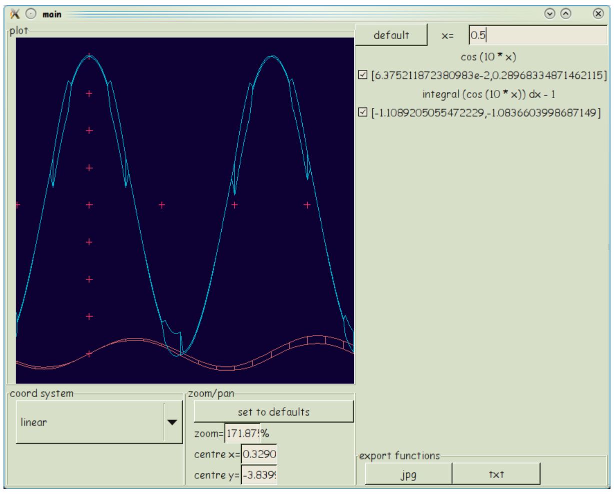
8.3 Mathematical Objects

8.3.1 AERN-Real and Friends

Report by:	Michal Konečný
Participants:	Amin Farjudian, Jan Duracz
Status:	experimental, actively developed

AERN stands for Approximating Exact Real Numbers. We are developing a family of the following libraries for fast exact real number arithmetic:

- AERN-Real: arbitrary precision safely rounded interval arithmetic with multiple backends (pure Haskell floating point numbers, MPFR, machine doubles) and with support for inner rounding, anti-consistent intervals and Kaucher arithmetic
- AERN-RnToRm: arbitrary precision safely-rounded arithmetic of piece-wise polynomial function enclosures (PFEs) for functions over n-dimensional real intervals with support for inner rounding, anti-consistent intervals and approximated Kaucher arithmetic
- AERN-RnToRm-Plot: GTK window for inspecting the graphs of PFEs in one variable (see figure below, showing a screenshot of an AERN-RnToRm-Plot window exploring an enclosure of $cos(10x)$ (blue) and an enclosure of its primitive function (red))
- AERN-Net: an implementation of distributed query-based (i.e., lazy) computation over analytical and geometrical objects



x= 0.5
cos (10 * x)
☑ [6.375211872380983e-2,0.28968334871462115]
integral (cos (10 * x)) dx - 1
☑ [-1.1089205055472229,-1.0836603998687149]
coord system: linear
zoom= 171.875%
centre x= 0.3290
centre y= -3.839

The development has been driven mainly by the needs of our two recent research projects. We have used the libraries extensively to:

- prototype algorithms for reliable and ultimately converging methods for solving differential equations in many variables (AERN-RnToRm, AERN-Net)
- solve numerical constraint satisfaction problems, especially those arising from verification of programs that use floating point numbers (AERN-RnToRm)

The current versions have been fairly stable for our purposes.

We are currently redesigning and rewriting the libraries almost from scratch with the following goals:

- A larger number of simpler and more reusable type classes instead of the few and fairly complex type classes provided in the current version; this includes type classes such as RoundedLattice or RoundedMultiplication. (Mostly completed for real interval arithmetic.)

- A more thorough approach to testing, with properties defined alongside the type classes. (Mostly completed for real interval arithmetic.)
- Support for both pure arithmetic and in-place updates for extra efficiency with backends written in C such a MPFR. (In progress.)
- A faster implementation of polynomial arithmetic, with a core written in C. (In progress.)

Further reading

- See Haddock documentation via Hackage — has links to research papers.
- New version under construction on http://code.google.com/p/aern/.

8.3.2 hmatrix

Report by:	Alberto Ruiz
Participants:	Vivian McPhail
Status:	stable, maintained

`hmatrix` is a purely functional interface to numerical linear algebra, internally implemented using GSL, BLAS, and LAPACK.

Version 0.10 has been recently released to Hackage. It includes support for `Float` and `Complex Float` elements (excluding LAPACK computations), `Binary` instances, and new functions like monadic map for vectors and matrix product with optimum association order.

I thank Vivian McPhail for many excellent contributions.

Further reading

http://code.haskell.org/hmatrix

8.4 Data Types and Data Structures

8.4.1 HList — A Library for Typed Heterogeneous Collections

Report by:	Oleg Kiselyov
Participants:	Ralf Lämmel, Keean Schupke, Gwern Branwen

HList is a comprehensive, general purpose Haskell library for typed heterogeneous collections including extensible polymorphic records and variants. HList is analogous to the standard list library, providing a host of various construction, look-up, filtering, and iteration primitives. In contrast to the regular lists, elements of heterogeneous lists do not have to have the same type. HList lets the user formulate statically checkable constraints: for example, no two elements of a collection may have the same type (so the elements can be unambiguously indexed by their type).

An immediate application of HLists is the implementation of open, extensible records with first-class, reusable, and compile-time only labels. The dual application is extensible polymorphic variants (open unions). HList contains several implementations of open records, including records as sequences of field values, where the type of each field is annotated with its phantom label. We, and now others (Alexandra Silva, Joost Visser: PURe.CoddFish project), have also used HList for type-safe database access in Haskell. HList-based Records form the basis of OOHaskell (http://code.haskell.org/OOHaskell). The HList library relies on common extensions of Haskell 2010.

HList is being used in AspectAG (→5.4.2), typed EDSL of attribute grammars, and in HaskellDB. There has been many miscellaneous changes related to the names of exposed modules, fixity declarations. Patches by Adam Vogt significantly improve the Haddock-generated documentation. The current version is 0.2.3; it works with GHC 6.12.

We are investigating the use of type functions provided in the new versions of GHC.

Further reading

- HList: http://homepages.cwi.nl/~ralf/HList/
- OOHaskell: http://homepages.cwi.nl/~ralf/OOHaskell/

8.4.2 Verified Priority Queues

Report by:	Jim Apple
Status:	stable

A priority queue (or sometimes “heap”) is a container supporting the insertion of elements and the extraction of the minimum element. Gerth Brodal and Chris Okasaki presented a purely functional priority queue that also supports an $O(1)$ meld operation in their paper “Optimal Purely Functional Priority Queues”. This project provides an implementation of these priority queues that has been verified using the Coq proof assistant.

It is available on Hackage and can be installed with `cabal install meldable-heap`. The Coq proofs are included in the package.

Further reading

http://hackage.haskell.org/package/meldable-heap/

8.4.3 Graphalyze

Report by:	Ivan Lazar Miljenovic
Status:	version 0.10.0.1

See: http://haskell.org/communities/11-2009/html/report.html#sect5.7.2.

8.5 Generic and Type-Level Programming

8.5.1 FlexiWrap

Report by:	Iain Alexander
Status:	forthcoming

A library of flexible newtype wrappers which simplify the process of selecting appropriate typeclass instances, which is particularly useful for composed types.

A proof-of-concept prototype exists, and is being packaged for release on Hackage. A document describing the concept is planned.

8.5.2 uniplate

Report by:	Neil Mitchell

Uniplate is a library for writing simple and concise generic operations. Uniplate has similar goals to the original Scrap Your Boilerplate work, but is substantially simpler and faster. If you are writing any sort of compiler, you should be using a generics library. If you do not know any generics libraries, Uniplate is a good place to start.

Uniplate has recently undergone major revisions. The new version drops Haskell 98 compatibility, in favor of Haskell 2010 compatibility — simplifying the module layout. All the instances have been revised with a focus on performance. Some of the instances can now be generated by the Derive tool (→ 6.4.2). The instances based on the Data class have been optimized and extended — they now work on more types, and run faster.

Further reading

http://community.haskell.org/~ndm/uniplate/

8.5.3 Generic Programming at Utrecht University

Report by:	José Pedro Magalhães
Participants:	Johan Jeuring, Sean Leather, Andres Löh, Thomas van Noort, Martijn van Steenbergen, Sebastiaan Visser
Status:	actively developed

One of the research themes investigated within the Software Technology Center in the Department of Information and Computing Sciences at Utrecht University is generic programming. Over the last 10 years, we have played a central role in the development of generic programming techniques, languages, and libraries.

Currently we maintain a number of generic programming libraries and applications. We report most of them in this entry; our new library for generic deriving (→ 6.4.1) has its own entry, and `emgm` was reported on before (http://haskell.org/communities/05-2009/html/report.html#sect5.9.3).

`instant-generics` Using type families and type classes in a way similar to `multirec` and `regular`, `instant-generics` is yet another approach to generic programming, supporting a large variety of datatypes and allowing the definition of type-indexed datatypes. It was first described by Chakravarty et al., and forms the basis of one of our rewriting libaries.

`multirec` This library represents datatypes uniformly and grants access to sums (the choice between constructors), products (the sequence of constructor arguments), and recursive positions. Families of mutually recursive datatypes are supported. Functions such as *map*, *fold*, *show*, and equality are provided as examples within the library. Using the library functions on your own families of datatypes requires some boilerplate code in order to instantiate the framework, but is facilitated by the fact that `multirec` contains Template Haskell code that generates these instantiations automatically.

The `multirec` library can also be used for type-indexed datatypes. As a demonstration, the `zipper` library is available on Hackage. With this datatype-generic zipper, you can navigate values of several types.

Unfortunately, `multirec` does not work well with ghc-6.12, but it works fine in ghc-7 (thanks to the new type checker).

We are still planning to extend the `multirec` library with support for parameterized datatypes and datatype compositions.

`regular` While `multirec` focuses on support for mutually recursive regular datatypes, `regular` supports only single regular datatypes. The approach used is similar to that of `multirec`, namely using type families to encode the pattern functor of the datatype to represent generically. There have been no major releases of the `regular` or `regular-extras` packages on Hackage since the last report. The current versions provide a number of typical generic functions, but also some less well-known but useful functions: deep *seq*, `QuickCheck`'s *arbitrary* and *coarbitrary*, and `binary`'s *get* and *put*.

`syb` Scrap Your Boilerplate (`syb`) has been supported by GHC since the 6.0 release. This library is based on combinators and a few primitives for type-safe casting and processing constructor applications. It was originally developed by Ralf Lämmel and Simon Peyton Jones. Since then, many people have contributed with research relating to `syb` or its applications.

Since `syb` has been separated from the `base` package, it can now be updated independently of GHC. We have recently released version 0.2 on Hackage, which has reintegrated the testsuite and introduced new generic producers, along with smaller changes and fixes.

Annotations Recently we presented two applications of generic annotations at the Workshop on Generic Programming: selections and storage. In the former we use annotations at every recursive position of a datatype to allow for inserting position information automatically. This allows for informative parsing error messages without the need for explicitly changing the datatype to contain position information. In the latter we use the annotations as pointers to locations in the heap, allowing for transparent and efficient data structure persistency on disk.

Rewriting We also maintain two libraries for generic rewriting: a simple, earlier library based on `regular`, and the guarded rewriting library, based on `instant-generics`. The former allows for rewriting only on regular datatypes, while the latter supports more datatypes and also rewriting rules with preconditions.

We also continue to look at benchmarking and improving the performance of different libraries for generic programming (→ 8.5.4).

Further reading

http://www.cs.uu.nl/wiki/GenericProgramming

8.5.4 Optimizing Generic Functions

Report by:	José Pedro Magalhães
Participants:	Johan Jeuring, Andres Löh
Status:	actively developed

Datatype-generic programming increases program reliability by reducing code duplication and enhancing reusability and modularity. Several generic programming libraries for Haskell have been developed in the past few years. These libraries have been compared in detail with respect to expressiveness, extensibility, typing issues, etc., but performance comparisons have been brief, limited, and preliminary. It is widely believed that generic programs run slower than hand-written code.

At Utrecht University we are looking into the performance of different generic programming libraries and how to optimize them. We have confirmed that generic programs, when compiled with the standard optimization flags of the Glasgow Haskell Compiler (GHC), are substantially slower than their hand-written counterparts. However, we have also found that more advanced optimization capabilities of GHC can be used to further optimize generic functions, sometimes achieving the same efficiency as hand-written code.

We have benchmarked four generic programming libraries: `emgm`, `syb`, `multirec`, and `regular`. We compare different generic functions in each of these libraries to a hand-written version. We have concluded that inlining plays a crucial role in the optimization of generics. Previously we used flags to increase the chances of the GHC inliner to optimize our functions. However, such flags change the behavior of the inliner for the entire set of modules being compiled, which might have detrimental effects on performance. Currently we are investigating how to localize these hints to the compiler by using `INLINE` pragmas.

In most cases, we can achieve very good performance results by providing `INLINE` pragmas to the conversion functions (*from* and *to*) for each datatype and for each instance of the generic function on a representation type (such as *Sum*, *Prod*, etc.). We have to be careful with the optimization phases, as sometimes inlining too early can prevent later optimizations. In this way, we achieve the same performance as a type-specific hand-written version for functions like *show* and *update*, using only the infrastructure that GHC already provides. The performance of generic *read* is also significantly improved.

Unfortunately, some generic functions are still difficult to optimize with this technique. In particular, functions which involve additional datatypes in their type (such as *enum*, which returns a list of elements) prevent proper optimization. We are currently looking into how we can circumvent this restriction. We also plan to update our libraries to add the necessary pragmas for increased efficiency, but since we require the new inliner we have to wait until GHC version 7 is released.

Further reading

http://dreixel.net/research/pdf/ogie.pdf

8.6 User Interfaces

8.6.1 Gtk2Hs

Report by:	Axel Simon
Participants:	Andy Stewart and many others
Status:	beta, actively developed

Gtk2Hs is a set of Haskell bindings to many of the libraries included in the Gtk+/Gnome platform. Gtk+ is an extensive and mature multi-platform toolkit for creating graphical user interfaces.

GUIs written using Gtk2Hs use themes to resemble the native look on Windows. Gtk is the toolkit used by Gnome, one of the two major GUI toolkits on Linux. On Mac OS programs written using Gtk2Hs are run by Apple's X11 server but may also be linked against a native Aqua implementation of Gtk.

Gtk2Hs features:

- ○ Automatic memory management (unlike some other C/C++ GUI libraries, Gtk+ provides proper support for garbage-collected languages)
- ○ Unicode support
- ○ High quality vector graphics using Cairo
- ○ Extensive reference documentation
- ○ An implementation of the "Haskell School of Expression" graphics API
- ○ Bindings to many other libraries that build on Gtk: gio, GConf, GtkSourceView 2.0, glade, gstreamer, vte, webkit

The release of Gtk2Hs as a set of cabal packages has markedly increased the interest in the binding. We have been able to incorporate several bug fixes that were still lurking in the multi-threading support of our memory management. Another oversight were some lacking functions in the cairo and pango base packages that precluded their use to generate, e.g., PDF files without linking to the gtk package. We released version 0.11.2 in August that fixed both issues. Andy Steward has since been working hard to add many more functions, getting Gtk2Hs very close to support all features of the latest Gtk+ versions. John Obbele has kindly investigated into fixing some memory leaks. We will therefore push out a new version shortly.

Gtk2Hs version 0.11.2 has been released on August 15th. The next version will follow soon.

Further reading

- ○ News, downloads, and documentation: http://haskell.org/gtk2hs/
- ○ Development version: `darcs get` http://code.haskell.org/gtk2hs/

8.6.2 Haskeline

Report by:	Judah Jacobson
Status:	active development

The Haskeline library provides a user interface for line input in command-line programs. It is similar in purpose to readline or editline, but is written in Haskell and aims to be more easily integrated into other Haskell programs. A simple, monadic API allows this library to provide guarantees such as restoration of the terminal settings on exit and responsiveness to control-c events.

Haskeline supports Unicode and runs both on the native Windows console and on POSIX-compatible systems. It has a rich, user-customizable line-editing interface. Recent improvements include support for languages with wide characters and several optimizations for speed and responsiveness. Additionally, the API now provides hidden password entry and allows more control over the choice between terminal-style and file-style interactions.

Further reading

- ○ http://trac.haskell.org/haskeline
- ○ http://hackage.haskell.org/package/haskeline

8.6.3 CmdArgs

Report by:	Neil Mitchell
Status:	released

CmdArgs is a library for defining and parsing command lines. The focus of CmdArgs is allowing the concise definition of fully-featured command line argument processors, in a mainly declarative manner (i.e., little coding needed). Compared to the standard GetOpt library, CmdArgs is often about three times shorter. CmdArgs also supports multiple mode programs, for example as used in git/darcs/Cabal.

Further reading

http://community.haskell.org/~ndm/cmdargs/

8.7 Graphics

8.7.1 plot/plot-gtk

Report by:	Vivian McPhail
Status:	Active Development

`plot` is an embedded domain-specific language for the generation of figures. `plot-gtk` is a driver that provides a GTK widget to display figures and also a wrapper that allows interactive plotting sessions with GHCi. The package generates instructions for the Cairo renderer, which can be used to output figures in PS, PDF, PNG, and SVG file formats.

The motivation for this package is to provide a tool both for publication quality graphics and for the interactive visualisation of mathematical objects as a Haskell replacement for octave/gnuplot, matlab, and other non-Haskell numerical tools.

Users can plot functions of type `Double -> Double` and data series of type `Vector Double`, which are compatible with the high-performance `vector` package when `hmatrix` (→ 8.3.2) is installed with the `-fvector` flag.

Features:

- ○ simple monadic interface to configure each figure and elements
- ○ title/subtitle
- ○ an array of plots in each figure with optional headers
- ○ configurable axes and ticks
- ○ configurable ranges

- linear, log, semilog ranges
- plot vectors or `(Double -> Double)` functions
- line/point/linepoint/impulse/step/area plots
- bar/histogram plots
- optional error bars
- mix and match data series types and formatting
- fully configurable text elements
- greyscale matrix visualisation

The `plot/plot-gtk` packages have just had their initial release and are available from Hackage.
Work is being done to:

- improve tick labelling and formatting
- reduce burden on the user for bar chart layout
- give elements layout tags for improved customisation
- extend the Simple interface

  3D plots are planned for a future release.

Further reading

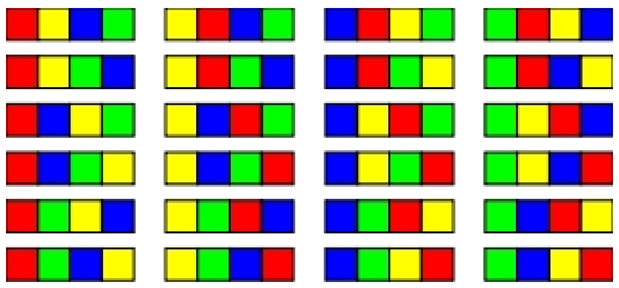
- http://hackage.haskell.org/package/plot
- http://hackage.haskell.org/package/plot-gtk

8.7.2 diagrams

Report by:	Brent Yorgey
Status:	active development

The diagrams library provides an embedded domain-specific language for creating simple pictures and diagrams. Values of type `Diagram` are built up in a compositional style from various primitives and combinators, and can be rendered to a physical medium, such as a file in PNG, PS, PDF, or SVG format. The overall vision is for diagrams to become a viable alternative to DSLs like MetaPost or Asymptote, but with the advantages of being *purely functional* and *embedded*.

For example, consider the following diagram to illustrate the 24 permutations of four objects:



The diagrams library was used to create this diagram with very little effort (about ten lines of Haskell, including the code to actually generate permutations). The source code for this diagram, as well as other examples and further resources, can be found at http://code.haskell.org/diagrams/.

The library is currently undergoing a major rewrite, with the goal of basing the entire library on a more flexible, semantically elegant foundational core. Good progress was made at the most recent Philadelphia hackathon, and a preliminary release is in the works. Planned features include pluggable rendering backends, support for arbitrary vector spaces and for animation, more sophisticated paths and path operations, and an xmonad-like core/contrib model for incorporating user-submitted extension modules.

Further reading

- http://code.haskell.org/diagrams/
- http://byorgey.wordpress.com/2009/09/24/diagrams-0-2-1-and-future-plans/
- http://www.tug.org/metapost.html
- http://asymptote.sourceforge.net/

8.7.3 GPipe

Report by:	Tobias Bexelius

GPipe models the entire graphics pipeline in a purely functional, immutable and type-safe way. It is built on top of the programmable pipeline (i.e., non-fixed function) of OpenGL 2.1 and uses features such as vertex buffer objects (VBO's), texture objects, and GLSL shader code synthetization to create fast graphics programs. Buffers, textures, and shaders are cached internally to ensure fast framerate, and GPipe is also capable of managing multiple windows and contexts. GPipe's aim is to be as close to the conceptual graphics pipeline as possible, and not to add any more levels of abstraction.

In GPipe, you work with four main data types: PrimitiveStreams, FragmentStreams, FrameBuffers, and textures. They are all immutable, and all parameterized on the type of data they contain to ensure type safety between pipeline stages. By creating your own instances of GPipes type classes, it is possible to use additional data types on the GPU.

Version 1.2.1 with documentation is released on Hackage, as well as some utility libraries that enable loading of Collada geometries and JPEG textures. There are also a few examples and tutorials that can be found through the wiki.

I am not currently working on any more additions myself, but the sources are available on github and anyone is welcome to contribute.

Further reading

http://www.haskell.org/haskellwiki/GPipe

8.7.4 ChalkBoard

Report by:	Andy Gill
Participants:	Kevin Matlage
Status:	ongoing

ChalkBoard is a domain specific language for describing images. The language is uncompromisingly functional and encourages the use of modern functional idioms. The novel contribution of ChalkBoard is that it uses off-the-shelf graphics cards to speed up rendering of our functional description. We always intended to use ChalkBoard to animate educational videos, as well as for processing streaming videos. Since the last HCAR report, we have added a new animation language, based round a new applicative functor, `Active`. It has been called Functional Reactive Programming, without the reactive part! The paper "Every Animation Should Have a Beginning, a Middle, and an End" talks about this addition.

A release is scheduled for November 2010.

Further reading

http://www.ittc.ku.edu/csdl/fpg/Tools/ChalkBoard

8.7.5 graphviz

Report by:	Ivan Lazar Miljenovic
Status:	version 2999.10.0.1

The graphviz library provides Haskell bindings for the *Graphviz* suite of tools for visualizing graphs by utilizing Graphviz's *Dot* language. The major features of the graphviz library include:

- Almost complete coverage of all Graphviz attributes and syntax.
- Support for specifying clusters.
- The ability to use a custom node type.
- Functions for running a Graphviz layout tool with all specified output types.
- The ability to not only generate but also parse Dot code with two options: strict and liberal (in terms of ordering of statements).
- Functions to convert FGL graphs to Dot code — including support to group them into clusters — with a high degree of customization by specifying which attributes to use and limited support for the inverse operation.
- Round-trip support for passing an FGL graph through Graphviz to augment node and edge labels with positional information, etc.

For a sample graph visualized using the graphviz library, see SourceGraph (→ 6.3.2).

Further reading

- http://projects.haskell.org/graphviz/
- http://hackage.haskell.org/package/graphviz
- http://www.graphviz.org/

8.8 Text and Markup Languages

8.8.1 HaTeX

Report by:	Daniel Díaz
Status:	active development

HaTeX is a library with the purpose of providing the possibility of integrating the script of a LaTeX file in a program written in Haskell. The integration takes place through the well known monadic transformer `WriterT`, which stores in its state the LaTeX code. The library provides a set of functions for adding the code, and you can include your monadic computations making use of a lifting function. HaTeX is really easy to use if you know LaTeX already, and only a little effort is enough otherwise. The documentation will help to learn to utilize and understand this library, with the initial guide (to be found in HaskellWiki), the extended guide "*HaTeX, a monadic perspective of LaTeX*", or the API documentation. The latter is not completed yet, due to the large number of entities. But, if you know LaTeX, you can help to solve this.

Further reading

http://ddiaz.asofilak.es/packages/HaTeX

8.8.2 Haskell XML Toolbox

Report by:	Uwe Schmidt
Status:	seventh major release (current release: 9.0)

Description

The Haskell XML Toolbox (HXT) is a collection of tools for processing XML with Haskell. It is itself purely written in Haskell 98. 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). There is a validator based on DTDs and a new more powerful one for Relax NG schemas.

The Haskell XML Toolbox is based on the ideas of HaXml and HXML, but introduces a more general approach for processing XML with Haskell. The processing model is based on arrows. The arrow interface is more flexible than the filter approach taken in the earlier HXT versions and in HaXml. It is also safer; type checking of combinators becomes possible with the arrow approach.

HXT is partitioned into a collection of smaller packages: The core package is `hxt`. It contains a validating XML parser, an HTML parser, filters for manipulating XML/HTML and so called XML pickler for converting XML to and from native Haskell data.

Basic functionality for character handling and decoding is separated into the packages hxt-charproperties and hxt-unicode. These packages may be generally useful even for non XML projects.

HTTP access can be done with the help of the packages hxt-http for native Haskell HTTP access and hxt-curl via a libcurl binding. An alternative lazy non validating parser for XML and HTML can be found in hxt-tagsoup.

The XPath interpreter is in package hxt-xpath, the XSLT part in hxt-xslt and the Relax NG validator in hxt-relaxng. For checking the XML Schema Datatype definitions, also used with Relax NG, there is a separate and generally useful regex package hxt-regex-xmlschema.

The old HXT approach working with filter hxt-filter is still available, but currently only with hxt-8. It has not (yet) been updated to the hxt-9 mayor version.

Features

- Validating XML parser
- Very liberal HTML parser
- Lightweight lazy parser for XML/HTML based on Tagsoup (→ 8.8.3)
- Easy de-/serialization between native Haskell data and XML by pickler and pickler combinators
- XPath support
- Full Unicode support
- Support for XML namespaces
- Cabal package support for GHC
- HTTP access via Haskell bindings to libcurl and via Haskell HTTP package
- Tested with W3C XML validation suite
- Example programs
- Relax NG schema validator
- Lightweight regex library with full support of Unicode and XML Schema Datatype regular expression syntax
- An HXT Cookbook for using the toolbox and the arrow interface
- Basic XSLT support
- GitHub repository with current development versions of all packages http://github.com/UweSchmidt/hxt

Current Work

HXT has grown over the years. To make the toolbox more modular and to reduce the dependencies on other packages, hxt has been split into various smaller packages since version 9.0.0.

This enables bindings to other useful XML components. There is a binding to the hexpat package under construction for lazy and fast parsing via the expat parser implemented in C.

There are some plans to further develop the Relax NG validator for full XML Schema Datatype support and for the native Relax NG schema notation.

Further reading

The Haskell XML Toolbox Web page (http://www.fh-wedel.de/~si/HXmlToolbox/index.html) includes links to downloads, documentation, and further information.

A getting started tutorial about HXT is available in the Haskell Wiki (http://www.haskell.org/haskellwiki/HXT). The conversion between XML and native Haskell data types is described in another Wiki page (http://www.haskell.org/haskellwiki/HXT/Conversion_of_Haskell_data_from/to_XML).

8.8.3 tagsoup

Report by:	Neil Mitchell

TagSoup is a library for extracting information out of unstructured HTML code, sometimes known as tag-soup. The HTML does not have to be well formed, or render properly within any particular framework. This library is for situations where the author of the HTML is not cooperating with the person trying to extract the information, but is also not trying to hide the information.

The library provides a basic data type for a list of unstructured tags, a parser to convert HTML into this tag type, and useful functions and combinators for finding and extracting information. The library has seen real use in an application to give Hackage (→ 6.7.1) listings, and is used in Hoogle (http://haskell.org/communities/05-2009/html/report.html#sect4.4.1).

A new version of tagsoup has been released, fully supporting the HTML 5 specification. The API also has experimental support for ByteString (although currently ByteString is slower than String).

Further reading

http://community.haskell.org/~ndm/tagsoup

8.8.4 BlazeHtml

Report by:	Jasper Van der Jeugt
Participants:	Simon Meier
Status:	stable

Much has happened to the BlazeHtml project since the last HCAR. Thanks to Google Summer of Code, huge amounts of work have been done during the past summer. 0.1 and 0.2 versions were released on Hackage, documentation was written, and development is still continuing.

Perhaps the most interesting part of the project in the long term is the Blaze Builder, designed by Simon Meier. A lot of packages could benefit from this code, as it speeds up a number of really low-level actions. For example, it performs better than `Data.ByteString.pack` (a factor 2 for packing more than 250 bytes) and especially better than `Data.ByteString.Lazy.pack` (up to a factor 20 for 100kb bytestrings).

In this aspect, we hope the project has improved the state of HTML generation in Haskell, and perhaps even more than that…

Further reading

http://jaspervdj.be/blaze/

8.8.5 Bravo

Report by:	Matthias Reisner
Status:	experimental; active development

Bravo is a general-purpose text template library, providing the parsing and generation of templates at compile time. Templates can be read from strings or files and for each a new record data type is created, allowing convenient access to all template variables in a type-safe manner. The data type creation it achieved by the use of the Template Haskell language extension.

Features

Compared to other template libraries, Bravo's features are:

◦ Static template processing: All templates are read, parsed, and processed at compile time, so no extra file access or error handling at runtime is necessary.

◦ Multiple templates per file: Bravo allows the user to define multiple templates per file with arbitrary comments between them, e.g., for template documentation.

◦ Conditional template evaluation: To check conditions at runtime and return the appropriate template text, conditional template expressions are provided.

◦ Embedding of Haskell expressions: Bravo allows the user to embed arbitrary Haskell 98 expressions combined with template variables. The set of permitted functions/operators/types can be controlled using Haskell's module system.

◦ Customized data type generation: Bravo uses a default scheme for the data type creation that can be replaced by a user defined scheme easily.

Future plans

There were plans to extend Bravo's capabilities by introducing new template expressions, e.g., to "map" a template over a list of values. Contrary to expectations, this requires the internal parser to be rewritten and split into lexer and parser. However, this would also improve extensibility and stability of the implementation. Further work will include performance analysis and handling of different input encodings. Support for custom template expression delimiters (the current are `{{` and `}}`) and caching are also planned.

Further reading

◦ http://www.haskell.org/haskellwiki/Bravo
◦ http://hackage.haskell.org/package/Bravo

9 Applications and Projects

9.1 Education

9.1.1 Holmes, Plagiarism Detection for Haskell

Report by:	Jurriaan Hage
Participants:	Brian Vermeer

Holmes is a tool for detecting plagiarism in Haskell programs. It has been implemented by Brian Vermeer under supervision of Jurriaan Hage as a master thesis project. The idea was to discover what heuristics work well. We found that a token stream based and the fingerprinting of Moss work well enough, *if* you remove template code and dead code before the comparison. There is one exception: refactorings that introduce a small piece of code in many places. For example, adding debug statements all around will degrade scores quite fast, particularly when combined with other refactorings. We do have another control-flow graph based heuristics that seems to perform quite well in this case, and, as a sideline, we have a student in our department who has developed an algorithm for near graphisomorphism that seems to work really well in comparing control-flow graphs in an inexact fashion.

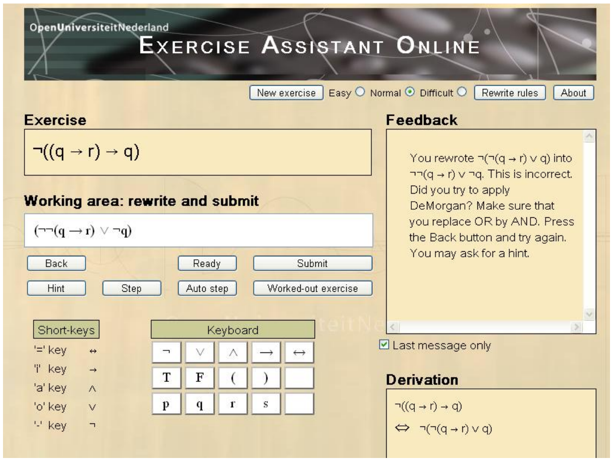
We have a prototype tool that works for Helium programs (→3.4), and we did some preliminary studies with live Helium programs to discover plagiarism.

Another student will soon attempt to take Holmes to full Haskell by building a similar system on top of `haskell-src-exts`. We can then also easily evaluate the tool for a large collection of programs students have submitted over the years. We hope to report on this work in the next Haskell Symposium. The tool will *not* be made available through Hackage, but will be available to lecturers on request.

9.1.2 Interactive Domain Reasoners (previously: Exercise Assistants)

Report by:	Bastiaan Heeren
Participants:	Alex Gerdes, Johan Jeuring, Josje Lodder
Status:	experimental, active development

The Ideas project (at Open Universiteit Nederland and Universiteit Utrecht) aims at developing interactive domain reasoners on various topics. These reasoners assist students in solving exercises incrementally by checking intermediate steps, providing feedback on how to continue, and detecting common mistakes. The reasoners are based on a strategy language, from which all feedback is derived automatically. The calculation of feedback is offered as a set of web services, enabling external (mathematical) learning environments to use our work. We currently have a binding with the Digital Mathematics Environment (DWO) of the Freudenthal Institute, the ActiveMath learning system (DFKI and Saarland University), and our own online exercise assistant that supports rewriting logical expressions into disjunctive normal form.



Exercise

¬((q → r) → q)

Working area: rewrite and submit

(¬¬(q → r) ∨ ¬q)

Feedback

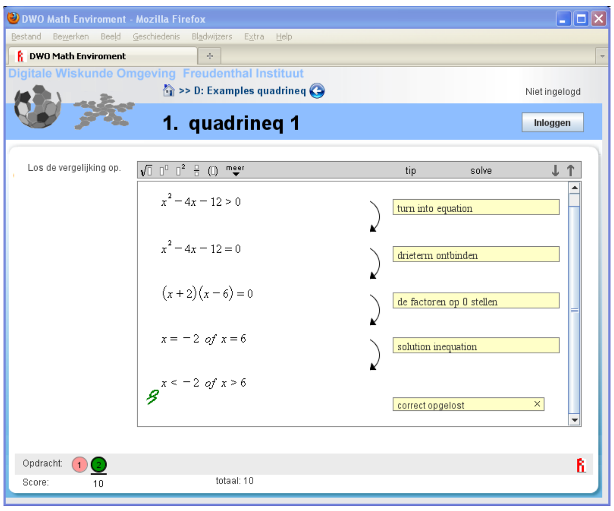
You rewrote ¬(¬(q → r) ∨ q) into ¬¬(q → r) ∨ ¬q. This is incorrect. Did you try to apply DeMorgan? Make sure that you replace OR by AND. Press the Back button and try again. You may ask for a hint.

'=' key	↔
'i' key	→
'a' key	∧
'o' key	∨
'-' key	¬

Derivation

¬((q → r) → q)

⇔ ¬(¬(q → r) ∨ q)


DWO Math Enviroment - Mozilla Firefox
Bestand Bewerken Beeld Geschiedenis Bladwijzers Extra Help
DWO Math Enviroment
Digitale Wiskunde Omgeving Freudenthal Instituut
>> D: Examples quadrineq
Niet ingelogd
Inloggen
meer
tip
solve

1. quadrineq 1

Los de vergelijking op.

$x^2 - 4x - 12 > 0$ turn into equation

$x^2 - 4x - 12 = 0$ drieterm ontbinden

$(x+2)(x-6) = 0$ de factoren op 0 stellen

$x = -2$ *of* $x = 6$ solution inequation

$x < -2$ *of* $x > 6$

correct opgelost

Opdracht: 1 2

Score: 10 totaal: 10

We are adding support for more exercise types, mainly at the level of high school mathematics. For example, our tool now covers simplifying expressions with exponents, rational equations, and derivatives. We have investigated how users can adapt mathematical domain reasoners to their own needs, such as the level of expertise. Recently, we have focused on designing a functional programming tutor. This tool lets you practice introductory functional programming exercises. We are also formalizing our strategy specification language, and the services that are derived from this language. This is ongoing research.

The feedback services are available as a Cabal source package.

Further reading

- Online exercise assistant (for logic), accessible from our project page.
- Bastiaan Heeren, Johan Jeuring and Alex Gerdes. Specifying Rewrite Strategies for Interactive Exercises. Mathematics in Computer Science, 3(3):349–370, 2010.
- Bastiaan Heeren, Johan Jeuring. Adapting Mathematical Domain Reasoners. International Conference on Mathematical Knowledge Management (MKM 2010).

9.1.3 Yahc

Report by:	Miguel Pagano
Participants:	Renato Cherini
Status:	testing, maintained

The first course on algorithms in CS at *Universidad Nacional de Córdoba* is centered on the derivations of algorithms from specifications, as proposed by R.S. Bird (*Introduction to functional programming using Haskell*, Prentice Hall Series in Computer Science, 1998), E.W. Dijkstra (*A Discipline of Programming*, Prentice Hall, 1976), and R.R. Hoogerwoord (*The design of functional programs: a calculational approach*, Technische Universiteit Eindhoven, 1989). To achieve this goal, students should acquire the ability to manipulate complex predicate formulae; thus the students first learn how to prove theorems in a propositional calculus similar to the equational propositional logic of D. Gries and F.B. Schneier (*A Logical Approach to Discrete Math*, Springer-Verlag, 1993).

During the semester students make many derivations as exercises and it is helpful for them to have a tool for checking the correctness of their solutions. Yahc checks the correctness of a sequence of applications of some axioms and theorems to the formulae students are trying to prove. The student starts a derivation by entering an initial formula and a goal and then proceeds by telling Yahc which axiom will be used and the expected outcome of applying the axiom as a rewrite rule; if that rewriting step is correct then the process continues until the student reaches the goal.

After the experience gained during one semester we made some changes in the user-interface. We have also added the definition of new constants and rules, which permits the resolution of logical puzzles.

In the long term we plan to consider an equational calculus with functions defined by induction over lists and natural numbers.

Further reading

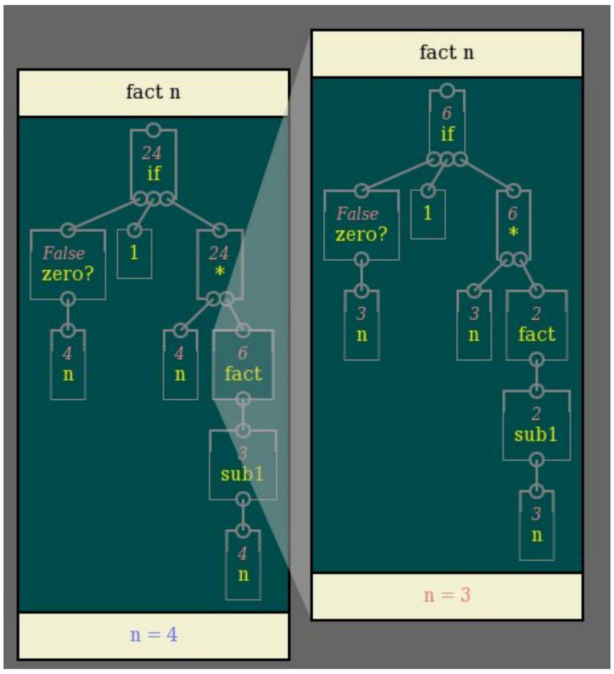
http://www.cs.famaf.unc.edu.ar/~mpagano/yahc/

9.1.4 Sifflet

Report by:	Gregory D. Weber
Status:	experimental, actively developed

Sifflet is a visual, functional programming language. Sifflet programmers define functions by drawing diagrams. Sifflet shows how a function call is evaluated on the diagram. It is intended as an aid for learning about recursion.

Here is Sifflet showing the first two levels of evaluating 4!:



Features

- Visual editor.
- Visual tracer/debugger which shows how function calls are evaluated, supporting an active learning process: Sifflet does not overwhelm students with a huge trace of function calls; it provides only as much expansion as the student requests.
- Extensive tutorial with 6,940 words and 31 pictures.
- Number, string, and list data types.
- A function “palette” with a small number of primitive functions.
- Runnable examples of compound functions.
- *New feature* (version 1.0, August 24, 2010): exports Haskell, Python 3, and Scheme code.
- *New feature* (version 1.2, October 29, 2010): Sifflet no longer indirectly depends on curl, which may make it easier for Windows users to install.

Availability

Sifflet made its public debut in May, 2010. It is available from Hackage: http://hackage.haskell.org/package/sifflet For Arch Linux, an AUR package is also available: http://aur.archlinux.org/packages.php?ID=39876

Future plans

In future releases, I hope to add these features:

- Type inference, and type declarations for exported Haskell code.
- Higher-order functions.
- Tree data and/or user-defined data types.

Further reading

- http://mypage.iu.edu/~gdweber/software/sifflet/home.html
- http://mypage.iu.edu/~gdweber/software/sifflet/doc/tutorial.html

9.2 Data Management and Visualization

9.2.1 HaskellDB

Report by:	Justin Bailey
Status:	active development

Scrap your SQL strings! The HaskellDB library provides a set of combinators based on the "relational algebra" for expressing queries, inserts, and updates. It lets you abstract over every part of your query, from conditions, to tables, to the columns returned. HaskellDB uses the HDBC family of database drivers to talk to a wide variety of databases.

Further reading

http://trac.haskell.org/haskelldb

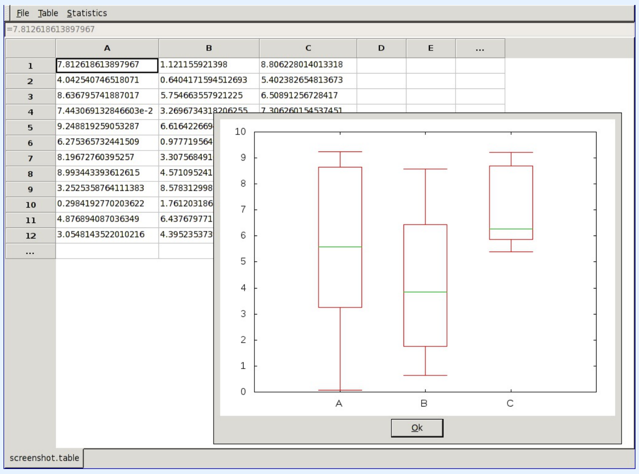
9.2.2 lhae

Report by:	Alexander Bau
Status:	in development, but stable

lhae is a simple spreadsheet application. It helps to manage your data in two-dimensional grids. Each cell in the grid contains a formula representing the stored information. Therefor lhae features a simple formula language: it supports various kinds of cell references, function calls, and conditional expressions. In order to provide automatic cell recalculation lhae keeps track of all cell dependencies.

lhae offers some table management operations like adding, deleting, inserting, transposing, and filtering of rows and columns. There are also some basic statistical methods like calculating frequency distributions, descriptive statistics, and pivot tables.

To integrate lhae in your flow of work you can import csv (character seperated values) files and export diagrams (using gnuplot).



	A	B	C	D	E	...
1	7.812618613897967	1.121155921398	8.806228014013318			
2	4.042540746518071	0.6404171594512693	5.402382654813673			
3	8.636795741887017	5.754663557921225	6.50891256728417			
4	7.443069132846603e-2	3.2696734318206255	7.306260154537451			
5	9.248819259053287	6.616422669				
6	6.275365732441509	0.977719564				
7	8.19672760395257	3.307568491				
8	8.993443393612615	4.571095241				
9	3.2525358764111383	8.578312998				
10	0.2984192770203622	1.761203186				
11	4.876894087036349	6.437679771				
12	3.0548143522010216	4.395235373				
...						

If you want to install lhae, you can use `cabal-install` by entering `cabal install lhae`.

The future development plans are mainly related to more advanced export features by supporting more of gnuplot's plotting qualities. But there also will be a more comprehensive set of functions the user can use in the formula language.

Further reading

http://www.imn.htwk-leipzig.de/~abau/lhae/

9.2.3 Pandoc

Report by:	John MacFarlane
Participants:	Andrea Rossato, Peter Wang, Paulo Tanimoto, Eric Kow, Luke Plant, Justin Bogner
Status:	active development

Pandoc aspires to be the swiss army knife of text markup formats: it can read markdown and (with some limitations) HTML, LaTeX, and reStructuredText, and it can write markdown, reStructuredText, HTML, DocBook XML, OpenDocument XML, ODT, RTF, groff man, MediaWiki markup, GNU Texinfo, LaTeX, ConTeXt, EPUB, Slidy, and S5. Pandoc's markdown syntax includes extensions for LaTeX math, tables, definition lists, footnotes, and more.

Since the last report, two new output formats have been added: EPUB and Slidy HTML slide shows. Now it is possible to write a book in markdown and produce an ebook with a single command! New markdown extensions include grid tables and example lists that are sequentially numbered throughout a document.

Further reading

http://johnmacfarlane.net/pandoc/

9.2.4 Ferry (Database-Supported Program Execution)

Report by:	Torsten Grust
Participants:	George Giorgidze, Tom Schreiber, Jeroen Weijers
Status:	active development

With project *Ferry* we try to establish a connection between two somewhat distant shores: programming languages and database technology. Ferry explores how far we can push the idea of relational database engines that directly and seamlessly participate in program evaluation to support the super-fast execution of data-intensive programs written in a variety of (functional) programming languages. Relational database systems (RDBMSs) provide the best understood and most carefully engineered query processing infrastructure available today. Notwithstanding these data processing capabilities, RDBMSs are often operated as plain stores that do little more than reproduce stored data items for further processing outside the database host. With Ferry, instead, we aim to turn the database system into an efficient, capable, and highly scalable co-processor for your programming language's runtime. To this end, we search for, design, and implement new compilation strategies that map data types (e.g., nested and ordered lists, arrays, dictionaries), control structures (e.g., nested iteration, conditionals, variable assignment and reference), and idioms prevalent in functional programming and scripting languages into efficient database queries. Here, we try to push the limits of what has been considered possible (this includes algebraic data types, pattern matching, higher-order functions, and closures, to name a few).

Variants of the Ferry technology have been used

- to enhance the SQL code generator in Philip Wadler's *Links*, such that a significantly larger class of Links programs may be considered *databaseable* now, and
- to create a capable and efficient version of *LINQ to SQL* provider (plugging into the Microsoft .NET Language Integrated Query framework).
- to create an integrated query facility for Haskell, called *Database supported Haskell* (DSH).

We are currently re-implementing the Ferry compiler in Haskell (using GHC). The Ferry compiler is used for both the independent Ferry language and as part of the embedded query language DSH for Haskell. DSH is suited to handle large scale data (e.g., social networks) in Haskell programs with familiar Haskell syntax. Both the compiler and DSH library will be published as an open source project soon.

Future plans

Ferry employs a compilation strategy revolving around the concept of *loop lifting* that appears to have quite close and interesting connections to the *flattening transformation* employed by Data Parallel Haskell. Indeed, Ferry understands the relational query engine as being a specific kind of data-parallel machine. The exact connection between Ferry and Data Parallel Haskell remains to be explored.

Further reading

http://www.ferry-lang.org

9.2.5 Sirenial

Report by:	Martijn van Steenbergen

Sirenial is an embedded DSL for modelling SQL statements.

```
select t0.townName
from towns t0
where (t0.id = 1)
```

The above query is the result of executing *getTownName* 1, where *getTownName* is defined as follows:

```
getTownName :: Ref Db.Town → Query String
getTownName townId = do
  [townName] ← select $ do
    t ← from Db.tableTown
    restrict (t # Db.townId . == . expr townId)
    return (t # Db.townName)
  return townName
```

The inner **do**-block is code in the *Select* monad, containing functions such as *from* and *restrict*, responsible for the creation of a single SELECT statement. These *Select* computations are lifted into the *Query* monad using *select*, where it becomes apparent that a query always yields a list of results.

The symbols prefixed with *Db* model the database schema:

```
data Town
tableTown = Table "towns"
          :: Table Town
townName  = Field tableTown "townName"
          :: Field Town      String
townId    = Field tableTown "id"
          :: Field Town      (Ref Town)
```

An unusual feature of Sirenial is the automatic combining of queries: if similar *Query* computations are composed in applicative fashion, Sirenial will merge them transparently and send only a single statement to the database server. For example, executing *for* [11..20] *getTownName* results in only one SELECT statement being sent:

```
select t0.id, t0.townName
from towns t0
where t0.id in (11,12,13,14,15,16,17,18,19,20)
```

By locally replaying the WHERE clauses, the results from the database are distributed over the original *select* calls. All this happens transparently: the user can write the queries as if they were executed one by one.

The library is designed to be *moderately type-safe*: catch many mistakes at compile-time, yet use simple types, leading to simple and understandable type errors. It is easy to predict the generated SQL, as there is very little rewriting done on the statements. Finally, Sirenial is designed in such a way that it is easy to switch to using the library completely or partially, on existing databases and existing data.

Further reading

http://code.google.com/p/sirenial/

9.2.6 The Proxima 2.0 Generic Editor

Report by:	Martijn Schrage
Participants:	Lambert Meertens, Doaitse Swierstra
Status:	actively developed

Proxima 2.0 is an open-source web-based version of the Proxima generic presentation-oriented editor for structured documents. The system is being maintained by Oblomov Systems (→ 10.6).

- Proxima is a *generic* editor. This means that the editor can be instantiated for arbitrary document types, supplemented by parser and presentation sheets. The content of a Proxima document can be mixed text, images and diagrams.
- Proxima is a *presentation-oriented editor*. This means that the user performs edit operations on the WYSIWYG presentation of the document.
- Proxima is aware of the structure of the document. While editing the presentation of the document, the edit operations may also be structural. For example, a section can be changed into a subsection.

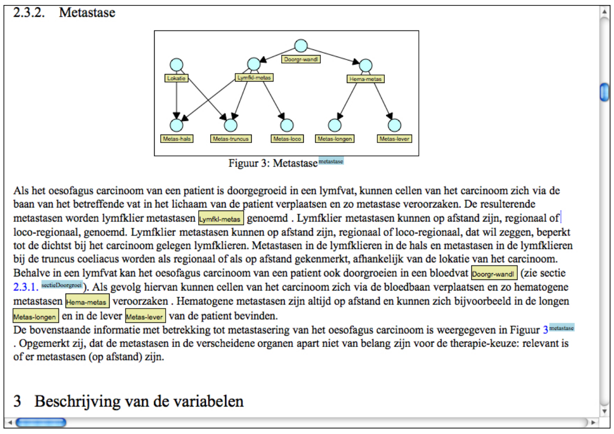
Another feature of Proxima is that it offers generic support for specifying content-dependent computations. For example, it is possible to create a table of contents of a document that is automatically updated as chapters or sections are added or modified.

Proxima 2.0

Proxima 2.0 provides a web-interface for Proxima. Instead of an application that renders onto a window, Proxima 2.0 is a web server that sends an HTML rendering of the document to a client. The client catches mouse and keyboard events, and sends these back to the server, after which the server sends an incremental rendering update back to the client. As a result, advanced editors can be created, which run in any browser. Among the current features of the system are drag and drop editing, session handling, and complex graphical presentations that may contain computed values and structures.

Because the (possibly large) HTML rendering may need to be communicated to the client on each key stroke or mouse gesture, Proxima 2.0 employs a number of techniques to ensure the editors respond fast enough over a network connection. On the one hand, *low bandwidth* may cause delays when sending large HTML renderings to the client. This problem is handled by using *incremental algorithms* to only send those parts of the rendering that were changed. On the other hand, *network latency* may cause a delay between a user edit gesture and the update received from the server. This problem is handled by using *predictive rendering*, which means that the client shows the predicted effect of the edit operation, until the actual update from the server is received and applied. Though both techniques may fail for pathological cases, they work very well for the majority of editors. As a result, the editors feel responsive enough even over remote network connections.

The Proxima website contains a gallery of live demo editors, as well as download instructions and documentation. The screenshot shows an editor for documenting Bayesian networks, running in Firefox.



2.3.2. Metastase

Figuur 3: Metastase metastase

Als het oesofagus carcinoom van een patient is doorgegroeid in een lymfvat, kunnen cellen van het carcinoom zich via de baan van het betreffende vat in het lichaam van de patient verplaatsen en zo metastase veroorzaken. De resulterende metastasen worden lymfklier metastasen Lymfkl-metas genoemd . Lymfklier metastasen kunnen op afstand zijn, regionaal of loco-regionaal, genoemd. Lymfklier metastasen kunnen op afstand zijn, regionaal of loco-regionaal, dat wil zeggen, beperkt tot de dichtst bij het carcinoom gelegen lymfklieren. Metastasen in de lymfklieren in de hals en metastasen in de lymfklieren bij de truncus coeliacus worden als regionaal of als op afstand gekenmerkt, afhankelijk van de lokatie van het carcinoom. Behalve in een lymfvat kan het oesofagus carcinoom van een patient ook doorgroeien in een bloedvat Doorgr-wandl (zie sectie 2.3.1. sectieDoorgroei). Als gevolg hiervan kunnen cellen van het carcinoom zich via de bloedbaan verplaatsen en zo hematogene metastasen Hema-metas veroorzaken . Hematogene metastasen zijn altijd op afstand en kunnen zich bijvoorbeeld in de longen Metas-longen en in de lever Metas-lever van de patient bevinden.
De bovenstaande informatie met betrekking tot metastasering van het oesofagus carcinoom is weergegeven in Figuur 3 metastase . Opgemerkt zij, dat de metastasen in de verscheidene organen apart niet van belang zijn voor de therapie-keuze: relevant is of er metastasen (op afstand) zijn.

3 Beschrijving van de variabelen

Future plans

Proxima 2.0 is an open source project. We are looking for people who would like to participate.

Further reading

- http://www.cs.uu.nl/wiki/bin/view/Proxima
- http://www.oblomov.com

9.3 Functional Reactive Programming

9.3.1 Functional Hybrid Modelling

Report by:	George Giorgidze
Participants:	Joey Capper, Henrik Nilsson
Status:	active research and development

The goal of the FHM project is to gain a better foundational understanding of non-causal, hybrid modelling and simulation languages for physical systems and ultimately to improve on their capabilities. At present, our central research vehicle to this end is the design and implementation of a new such language centred around a small set of core notions that capture the essence of the domain.

Causal modelling languages are closely related to synchronous data-flow languages. They model system behaviour using ordinary differential equations (ODEs) in explicit form. That is, cause-effect relationship between variables must be explicitly specified by the modeller. In contrast, non-causal languages model system behaviour using differential algebraic equations (DAEs) in implicit form, without specifying their causality. Inferring causality from usage context for simulation purposes is left to the compiler. The fact that the causality can be left implicit makes modelling in a non-causal language more declarative (the focus is on expressing the equations in a natural way, not on how to express them to enable simulation) and also makes the models much more reusable.

FHM is an approach to modelling which combines functional programming and non-causal modelling. In particular, the FHM approach proposes modelling with first class models (defined by continuous DAEs) using combinators for their composition and discrete switching. The discrete switching combinators enable modelling of hybrid systems (i.e., systems that exhibit both continuous and discrete dynamic behaviour). The key concepts of FHM originate from work on Functional Reactive Programming (FRP).

We are implementing Hydra, an FHM language, as a domain-specific language embedded in Haskell. The method of embedding employs quasiquoting and enables modellers to use the domain specific syntax in their models. The present prototype implementation of Hydra enables modelling with first class models and supports combinators for their composition and discrete switching.

We implemented support for dynamic switching among models that are computed at the point when they are being "switched in". Models that are computed at run-time are just-in-time (JIT) compiled to efficient machine code. This allows efficient simulation of highly structurally dynamic systems (i.e., systems where the number of structural configurations is large, unbounded or impossible to determine in advance). This goes beyond to what current state-of-the-art non-causal modelling languages can model. The implementation techniques that we developed should benefit other modelling and simulation languages as well.

We are also exploring ways of utilising the type system to provide stronger correctness guarantees and to provide more compile time reassurances that our system of equations is not unsolvable. Properties such as equational balance (ensuring that the number of equations and unknowns are balance) and ensuring the solvability of locally scoped variables are among our goals. Dependent types have been adopted as the tool for expressing these static guarantees. However, we believe that more practical type systems (such as system F) could be conservatively extended to make FHM safer without compromising their usability.

Recently, in an effort to showcase FHM and Hydra to the wider modelling and simulation community, we have modelled and simulated a number of challenging physical systems that current non-causal modelling languages can not handle (see the papers linked below).

Further reading

The implementation of Hydra and related papers are available from http://www.cs.nott.ac.uk/~ggg/.

9.3.2 Elerea

Report by:	Patai Gergely
Status:	experimental, active

Elerea (Eventless reactivity) is a tiny discrete time FRP implementation without the notion of event-based switching and sampling, with first-class signals (time-varying values). Reactivity is provided through various higher-order constructs that also allow the user to work with arbitrary time-varying structures containing live signals.

Stateful signals can be safely generated at any time through a specialised monad, while stateless combinators can be used in a purely applicative style. Elerea signals can be defined recursively, and external input is trivial to attach. The library comes in four major variants:

◦ `Simple`: signals are plain discrete streams isomorphic to functions over natural numbers;

◦ `Param`: adds a globally accessible input signal for convenience;

◦ `Clocked`: adds the ability to freeze whole subnetworks at will;

◦ `Delayed`: attempts to resolve instantaneous dependency cycles (i.e., cycles without a delay); this variant is likely to be deprecated in the near future due to its hairy semantics.

The first three variants come with precise denotational semantics.

The code is readily available via cabal-install in the `elerea` package. You are advised to install `elerea-examples` as well to get an idea how to build non-trivial systems with it. The examples are separated in order to minimize the dependencies of the core library. The experimental branch is showcased by Dungeons of Wor, found in the `dow` package (→5.3.3). Additionally, the basic idea behind the experimental branch is laid out in the WFLP 2010 article *Efficient and Compositional Higher-Order Streams.*

Since the last report, the library went through a major upgrade, during which the former experimental branch was promoted to be the primary interface, while the old version was tucked away in a legacy branch. Also, the `Clocked` branch is a recent addition.

Further reading

◦ http://hackage.haskell.org/package/elerea
◦ http://hackage.haskell.org/package/elerea-examples
◦ http://hackage.haskell.org/package/dow
◦ http://sgate.emt.bme.hu/documents/patai/publications/PataiWFLP2010.pdf
◦ http://babel.ls.fi.upm.es/events/wflp2010/video/video-08.html (WFLP talk)

9.4 Audio and Graphics

9.4.1 Audio Signal Processing

Report by:	Henning Thielemann
Status:	experimental, active development

In this project, audio signals are processed using pure Haskell code and the Numeric Prelude framework (http://haskell.org/communities/05-2009/html/report.html#sect5.6.2). The highlights are:

◦ a basic signal synthesis backend for Haskore (http://haskell.org/communities/05-2009/html/report.html#sect5.12.1),
◦ support for physical units while maintaining efficiency,
◦ frameworks for abstraction from sample rate, that is, the sampling rate can be omitted in most parts of a signal processing expression.
◦ We checked several low-level implementations in order to achieve reasonable speed. We complement the standard list type with a lazy `StorableVector` structure and a `StateT s Maybe a` generator, like in stream-fusion. Now, both our custom signal generator type and the Stream type from stream-fusion can be fused to work directly on storable vectors.
◦ support for causal processes. Causal signal processes only depend on current and past data and thus are suitable for real-time processing (in contrast to a function like time reversal). These processes are modeled as `mapAccumL` like functions. Many important operations like function composition maintain the causality property. They are important for sharing on a per sample basis and in feedback loops where they statically warrant that no future data is accessed.

Recent advances are:

◦ Lazy time values to be used for gate signals,
◦ enhanced type class framework for unifying lazy time values and signals expressed by lists, storable vectors or signal generators.
◦ Connection to alsa bindings, in order to provide real-time sound synthesis controlled by MIDI events from keyboards or sequencers,
◦ Stand-alone binding to Sox for audio format conversion and playback,
◦ A pyramid filter for efficient computation of moving average and moving maximum for baseline detection of mass spectra,
◦ A set of signal processors that generates maximally efficient vectorized code using LLVM as portable assembler.

Further reading

◦ http://www.haskell.org/haskellwiki/Synthesizer
◦ http://arxiv.org/abs/1004.4796

9.4.2 easyVision

Report by:	Alberto Ruiz
Status:	experimental, active development

The *easyVision* project is a collection of experimental libraries for computer vision and image processing. The low level computations are internally implemented by optimized libraries (IPP, HOpenGL, hmatrix (→ 8.3.2), etc.). Once appropriate geometric primitives have been extracted by the image processing wrappers we can define interesting computations using high level combinators.

Further reading

http://code.haskell.org/easyVision

9.4.3 n-Dimensional Volume Calculation for Non-Convex Polytops

Report by:	Farid Karimipour
Participants:	Mahmoud R. Delavar, Andrew U. Frank
Status:	active development

This is the continuation of the work "n-dimensional convex decomposition of polytops" (http://haskell.org/communities/11-2009/html/report.html#sect6.6.4) where we showed how to decompose an n-dimensional non-convex polytop to a set of convex components. The algorithm builds a tree of signed convex components that are stored as a set of n-simplexes: even levels are additive, whereas components in odd levels are subtractive. Here, the elements of this tree are utilized to calculate the volume of the original n-dimensional non-convex polytop ("volume" is used as a generalized term for all dimensions, i.e., "area" for 2D, etc.). The resultant components are triangulated whose volume calculation is straightforward:

$$V < (e_{01}, \ldots, e_{0n}), \ldots, (e_{n1}, \ldots, e_{nn}) > \; = \frac{1}{2} \begin{vmatrix} 1 & \ldots & 1 \\ e_{01} & \ldots & e_{n1} \\ \ldots & \ldots & \ldots \\ e_{0n} & \ldots & e_{nn} \end{vmatrix}$$

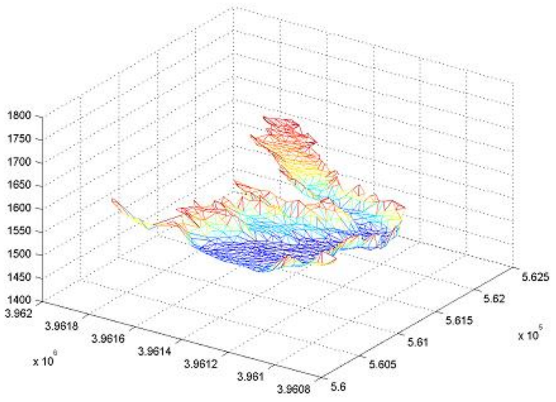
Summing up the volumes of all triangles (tetrahedrons in 3D) will provide us with the volume of the n-dimensional non-convex polytop:

$$V(P) = \sum_{i=0}^{n} \sum_{j=0}^{m_i} (-1)^i V(P_{ij})$$

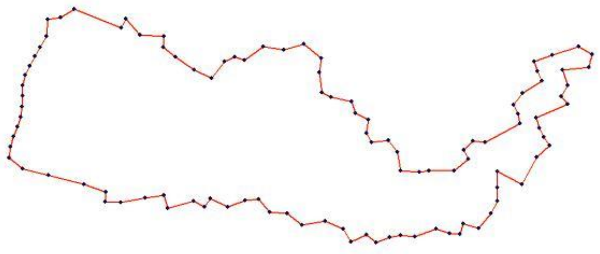
where P_{ij} means the jth component of the ith level and m_i is the number of components exist in the ith level. Note that this equation subtracts the volumes of the components of the odd levels. To implement this algorithm, the n-simplexes are represented as a list of points. Then, their operations (e.g., convex decomposition, triangulation, volume calculation, etc.) become operations on lists:

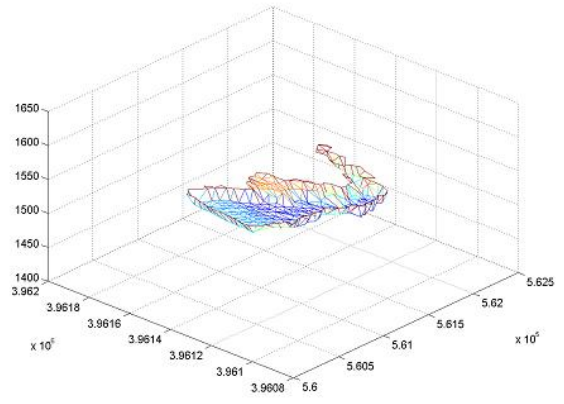
```
vS  = 0.5 (*) . abs . det . map (1 :)
vC  = sum . map vS . tri
vNC = sum (zipWith (*) (cycle [1, (−1)])
        (map vC cd))
```

where vS is the volume of an n-simplex, vC is the volume of a convex polytop, vNC is the volume of a non-convex polytop, tri is triangulation of a convex polytop and cd decomposes a non-convex polytop to a set of convex components. Since the representation and operations are defined independent of dimension, the developed algorithms can be used for polytops of any dimension.
The implementation was used to calculate the surface and volume of a lake at certain water levels, which leads to a level-surface-volume diagram. This diagram shows the surface and volume of the lake at different water levels. First, the 3D TIN (Triangulated Irregular Network) of the lake was constructed:

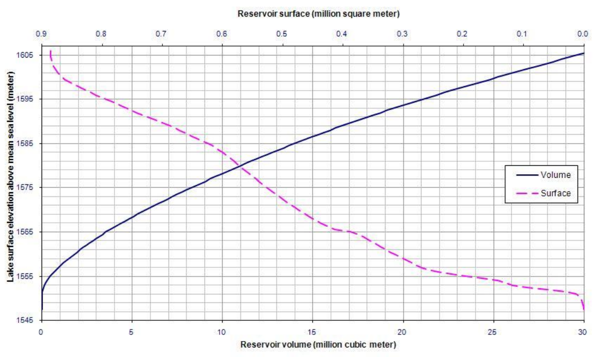


To calculate the surface and volume of the lake at a certain water level, say h, the 3D TIN was intersected with the plan $z = h$, which results in the volume of the lake where $z < h$ and the surface of the lake at $z = h$, whose surface and volume is calculated using the implemented algorithm:





By applying the above process for different water levels, the level-surface-volume diagram was produced:



9.4.4 Hemkay

Report by:	Patai Gergely
Status:	experimental, active

Hemkay (An M.K. Player Whose Name Starts with an H) is a simple music module player that performs all the mixing in Haskell. It supports the popular ProTracker format and some of its variations with different numbers of channels. The device independent mixing functionality can be found in the `hemkay-core` package.

The current version of the player uses the PortAudio bindings for playback, but there is also a yet unreleased functional version based on OpenAL, which puts a much smaller load on the CPU. Also, an OpenGL based graphical frontend is currently in the works.

Further reading

- http://hackage.haskell.org/package/hemkay-core
- http://hackage.haskell.org/package/hemkay
- http://en.wikipedia.org/wiki/MOD_(file_format)

9.5 Hardware Design

9.5.1 CλaSH

Report by:	Christiaan Baaij
Participants:	Matthijs Kooijman, Jan Kuper, Marco Gerards, Arjan Boeijink, Anja Niedermeier
Status:	experimental

CλaSH (CAES Language for Synchronous Hardware) is a functional hardware description language that borrows both its syntax and semantics from Haskell. The clock is implicit for the descriptions made in CλaSH: the behaviour of the circuit is described as transition from the current state to the next, which occurs every clock cycle. The current state is an input of such a transition function, and the updated state part of its result tuple. As descriptions are also valid Haskell, simulations can simply be performed by a Haskell compiler/interpreter (GHC only, due to the use of type families).

Instead of being an embedded language such as ForSyDe (→ 9.5.2) and Lava (→ 3.7)(→ 9.5.3)(→ 11.6), CλaSH has a compiler which can translate Haskell to synthesizable VHDL. The compiler has support for, amongst others: polymorphism, higher-order functions, user-defined abstract datatypes, and all of Haskell's choice mechanisms. The CλaSH compiler uses GHC for parsing, de-sugaring, and type-checking. The resulting Core-language description is then transformed into a normal form, from which a translation to VHDL is direct. The transformation system uses a set of rewrite rules which are exhaustively applied until a description is in normal form. Examples of these rewrite rules are β-reduction and η-expansion, but also transformations to transform higher-order functions to first-order functions, and transformation for the specialization of polymorphic functions.

The CλaSH compiler was first presented to the community, after 7 months of work, at the Haskell 2009 symposium in Edinburgh, Scotland. The compiler was stabilized in the following months, and two papers about CλaSH were accepted at more hardware-oriented conferences (DSD 2010 and FDL 2010). Support for arrows and the corresponding syntax, which eases the composition of transition functions, was added in July 2010 and was subsequently presented at IFL 2010 in Alphen a/d Rijn, The Netherlands.

The CλaSH compiler, available as a library, can be found both on Hackage (http://hackage.haskell.org/package/clash, stable) and github (http://github.com/christiaanb/clash/, development). The compiler/interpreter is also available as an executable, which is basically the GHC binary extended with the CλaSH library, on the CλaSH website (http://clash.ewi.utwente.nl).

Immediate plans for the future are to formalize the normalization system so completeness and termination can be proven. We also plan to add a Core-language evaluator so that the CλaSH compiler can use partial-evaluation techniques to support (finite) recursive descriptions and list constructs (instead of size-indexed vectors). This evaluator will also allow us to properly support recursively defined higher-order functions (e.g., `map`), instead of the current solution for these functions which is based on VHDL template code. There is also a design visualization tool in the making (expected January 2011).

Further reading

http://clash.ewi.utwente.nl

9.5.2 ForSyDe

Report by:	Ingo Sander
Participants:	Hosein Attarzadeh, Alfonso Acosta, Axel Jantsch, Jun Zhu
Status:	experimental

The ForSyDe (Formal System Design) methodology has been developed with the objective to move system-on-chip design to a higher level of abstraction. ForSyDe is implemented as a Haskell-embedded behavioral DSL.

ForSyDe allows to model heterogeneous embedded systems at a high level of abstraction by providing libraries for different models of computation (MoCs). This allows to model systems consisting of both digital and analog hardware.

The current release is ForSyDe 3.1, which contains two implementations of ForSyDe. The shallow-embedded DSL has been designed for the modeling purpose and provides a rapid-prototyping framework which allows to model and simulate heterogeneous embedded systems based on different MoCs. The deep-embedded DSL supports only the synchronous MoC, but comes with an embedded compiler with different backends (simulation, synthesizable VHDL and GraphML). It is possible to integrate and simulate shallow-embedded models with deep-embedded models.

The source code, together with example system models, is available from HackageDB under the BSD3 license.

Features

ForSyDe systems are modeled as concurrent process networks, where processes communicate via signals. To create processes, ForSyDe uses higher-order functions to implement the concept of process constructors, which leads to a structured model with a clear separation of computation from communication.

The two DSL flavors of ForSyDe offer different features:

1. Shallow-embedded DSL

   Shallow-embedded signals (`ForSyDe.Shallow.Signal`) are modeled as streams of data isomorphic to lists. Systems built with them are restricted to simulation. However, shallow-embedded signals provide a rapid-prototyping framework which allows to simulate heterogeneous systems based on different models of computation. At present ForSyDe supports the following models of computation.

   ◦ Synchronous MoC
   ◦ Untimed MoC
   ◦ Continuous Time MoC

   Process networks belonging to different MoCs communicate via domain interfaces, which establish a relation with respect to timing between two MoCs.

2. Deep-embedded DSL

   Deep-embedded signals (`ForSyDe.Signal`), based on the same concepts as Lava (→3.7)(→9.5.3)(→11.6), are aware of the system structure. Based on that structural information ForSyDe's embedded compiler can perform different analysis and transformations.

   ◦ Thanks to Template Haskell, specification of behavior is expressed in Haskell, not needing to specifically design a DSL for that purpose.
   ◦ Embedded compiler backends:
     – Simulation
     – VHDL (with support for Modelsim and Quartus II)
     – GraphML (with yFiles graphical markup support)
   ◦ Synchronous model of computation
   ◦ Support for hierarchy by component instantiation
   ◦ Support for fixed-sized vectors

ForSyDe allows to integrate deep-embedded models into shallow-embedded ones. This makes it possible to simulate a synthesizable deep-embedded model together with its environment, which may consist of analog and digital hardware, and software parts. Once the functionality of the deep-embedded model is validated, it can be synthesized to hardware using the VHDL-backend of ForSyDe's embedded compiler.

Further reading

http://www.ict.kth.se/forsyde/

9.5.3 Kansas Lava

Report by:	Andy Gill
Participants:	Tristan Bull, Andrew Farmer, Ed Komp
Status:	ongoing

Kansas Lava is a modern implementation of a hardware description language that uses functions to express hardware components, and leverages the abstractions in Haskell to build complex circuits. Lava, the given name for a family of Haskell based hardware description libraries (→3.7)(→11.6), is an idiomatic way of expressing hardware in Haskell which allows for simulation and synthesis to hardware.

Though there has been no public release (yet), we have made considerable progress with Kansas Lava. We have generated several large telemetry circuits, which have been synthesized and tested on real hardware, running at speeds comparable to other implementation techniques. A talk about internals of Kansas Lava was presented by Andrew Farmer at the Haskell implementors workshop in October, and the talk and slides are available online.

Jun Inoue from Rice University visited CSDL for October and November, to help connect his “staging” work with the Kansas Lava work.

A release of Kansas Lava release is planned for the end of the year.

Further reading

http://www.ittc.ku.edu/csdl/fpg/Tools/KansasLava

9.6 Proof Assistants and Reasoning

9.6.1 Zeno — Inductive Theorem Proving for Haskell Programs

Report by:	Will Sonnex
Participants:	Sophia Drossopoulou, Susan Eisenbach
Status:	Alpha 0.1.1

Zeno is a fully automated inductive theorem proving tool for proving properties of Haskell functions. You can express a property such as `takeWhile p xs ++ dropWhile p xs === xs` and it will prove it to be true for all values of `p :: a -> Bool` and `xs :: [a]`, over all types `a`, using only the function definitions.

After its most recent update Zeno can now reason about polymorphic types/functions, and you express the properties to be proven in Haskell itself (thanks to SPJ for the suggestion). It still cannot use all of Haskell's syntax: you cannot have internal functions (let/where can only assign values), and you cannot use type-classed polymorphic variables in function definitions — you will have to create a monomorphic instance of the function — but I hope to have these added reasonably soon. It is also still missing primitive-types/IO/imports so it still cannot be used with any real-world Haskell code, it is more a bit of theorem proving "fun".

Another feature is that Zeno lists all the sub-properties it has proven within each proof. When it verifies insertion-sort (`sorted (insertsort xs) === True`) it also proves the antisymmetry of `<=` and that the `insert` function preserves the `sorted` property.

You can try Zeno out at http://www.doc.ic.ac.uk/~ws506/tryzeno, the example code file given there has some provable properties about a few Prelude functions among other things. If you want the source code, it is available at http://code.google.com/p/zeno but I would advise you to use one of the branched versions, I make no guarantee that the trunk will even compile.

Further reading

http://www.doc.ic.ac.uk/~ws506/tryzeno

9.6.2 HTab

Report by:	Guillaume Hoffmann
Participants:	Carlos Areces, Daniel Gorin
Status:	active development
Current release:	1.5.4

HTab is an automated theorem prover for hybrid logics based on a tableau calculus. It handles hybrid logic with nominals, satisfaction operators, converse modalities, universal and difference modalities, the down-arrow binder, and role inclusion.

It is available on HackageDB (http://hackage.haskell.org/package/HTab) and comes with sample formulas to illustrate its input format.

The source code is distributed under the terms of the GNU GPL.

Further reading

http://code.google.com/p/intohylo/

9.6.3 Plastic

Report by:	Robin Adams
Participants:	Zhaohui Luo
Status:	prototype

The Plastic proof assistant was developed by Paul Callaghan in 2001 as an implementation of the logical framework LF, a Church-typed version of Martin-Löf's logical framework. Its development never advanced far beyond the experimental, prototype stage.

We have recently taken up the development of Plastic. A few years ago, Callaghan kindly adapted plastic to implement the Type Theoretic Framework, a framework for declaring several Logic-Enriched Type Theories (LETTs). We have already used this modified version of Plastic to formalize Weyl's *Das Kontinuum* in a classical predicative LETT.

We are currently experimenting with using Plastic for carrying out *pluralist formalizations*, where work in one mathematical setting may be reused in another setting, by providing an appropriate translation between the two.

For example, we have a proof script that proves a theorem A in a classical LETT. We may reuse this in a constructive LETT by plugging in a module that describes the double negation translation. The result is a proof of the double negation translation of A in the constructive LETT.

Plastic is written in Haskell.

Further reading

Details about this project will appear here soon: http://www.cs.rhul.ac.uk/~robin/plastic

9.6.4 Free Theorems for Haskell

Report by:	Janis Voigtländer
Participants:	Daniel Seidel, Matthias Bartsch, Joachim Breitner

Free theorems are statements about program behavior derived from (polymorphic) types. Their origin is the polymorphic lambda-calculus, but they have also been applied to programs in more realistic languages like Haskell. Since there is a semantic gap between the original calculus and modern functional languages, the underlying theory (of relational parametricity) needs to be refined and extended. We aim to provide such new theoretical foundations, as well as to apply the theoretical results to practical problems.

We maintain a library and tools for generating free theorems from Haskell types, originally implemented by Sascha Böhme and with contributions from Joachim Breitner and now Matthias Bartsch. Both the library and a shell-based tool are available from Hackage (as free-theorems and ftshell, respectively). There is also a web-based tool at http://www-ps.iai.uni-bonn.de/ft/. General features include:

- three different language subsets to choose from
- equational as well as inequational free theorems
- relational free theorems as well as specializations down to function level
- support for algebraic data types, type synonyms and renamings, type classes

The new version of the web-based tool, pepped up by Matthias, is now online. It enables the user to declare their own algebraic data types and so on, and then to derive free theorems from types involving those. (Previously, this was only possible in the shell-based tool.) Also, in addition to plain text, LaTeX source, and PDF output, the new version is able to output inline graphics with nicely typeset theorems. Matthias is now working on refactoring the internals of the generator, opening up possibilities for better control by the user, as well as for generating new forms of free theorems.

On the application side, we have another new web interface (http://www-ps.iai.uni-bonn.de/cgi-bin/b18n-combined-cgi), showcasing the technique (relying on free theorems for correctness) from ICFP'10 paper "Combining syntactic and semantic bidirectionalization".

Further reading

http://www.iai.uni-bonn.de/~jv/project/

9.6.5 Streaming Component Combinators

Report by:	Mario Blažević
Status:	experimental, actively developed

Streaming Component Combinators are an experiment at modeling dataflow architecture by using composable streaming components. All components are categorized into a small set of component types. A number of components can be composed into a compound component using a component combinator. For example, two transducer components can be composed together using a *pipe* operator into another transducer; one splitter and two transducers can be composed using an *if* combinator into a single compound transducer. Components are implemented as coroutines; the data flow among them is synchronous, but individual steps of different coroutines can run in parallel.

There are two ways to use SCC: as an embedded language in Haskell, or as a set of commands in a command-line shell. The latter provides its own parser and type checker, but otherwise relies on the former to do the real work.

The original work was done in the OmniMark programming language. Haskell was the language of choice for the second implementation because its strong typing automatically makes the embedded language strongly typed, and because its purity forces the implementation to expose the underlying semantics.

The currently planned future work includes extending the set of primitive components and component combinators and improving their performance, as well as extending the shell interface.

The latest stable version of SCC is available from Hackage.

Further reading

- Home page: http://trac.haskell.org/SCC/
- Hackage: http://hackage.haskell.org/package/scc
- Conference paper: Mario Blažević, Streaming component combinators, Extreme Markup Languages, 2006. http://www.idealliance.org/papers/extreme/proceedings/html/2006/Blazevic01/EML2006Blazevic01.html

- OmniMark implementation: http://developers.omnimark.com/etcetera/streaming-component-combinators.tar.gz

9.6.6 CSP-M Animator and Model Checker

Report by:	Marc Fontaine
Status:	active development, download available

We develop a Haskell based, integrated CSP-M animator and model checker.

Communicating-Sequential-Processes is a formalism for concurrent systems, invented by Tony Hoare.

Our Haskell-CSP-Tool features:

- FDR compatibility
- Fast computation of state spaces
- GTK+ based graphical user interface
- Support for shared-memory-parallelism / multicore CPUs

Binary releases of the gui-tool are available for download via http://www.stups.uni-duesseldorf.de/~fontaine/csp.

The aim of the project is not only to write a black-box end-user tool, but also to provide components that can be useful for other formal methods researchers who are investigating communicating sequential processes.

The following packages are available on Hackage:

CSPM-Frontend A FDR compatible CSP-M parser.

CSPM-CoreLanguage An abstract interface for a CSP core language.

CSPM-FiringRules An implementation of the firing rule semantic of CSP.

CSPM-Interpreter An interpreter for the functional sub language included in FDR.

CSPM-cspm A small command line executable that demonstrates how to clue the above libraries together.

Further reading

http://www.stups.uni-duesseldorf.de/~fontaine/csp

9.7 Natural Language Processing

9.7.1 NLP

Report by:	Eric Kow

See: http://haskell.org/communities/05-2009/html/report.html#sect6.10.1.

9.7.2 GenI

Report by:	Eric Kow

See: http://haskell.org/communities/11-2009/html/report.html#sect6.10.2.

9.7.3 Grammatical Framework

Report by:	Krasimir Angelov
Participants:	Olga Caprotti, Grégoire Détrez, Ramona Enache, Thomas Hallgren, Aarne Ranta

Grammatical Framework (GF) is a programming language for multilingual grammar applications. It can be used as a more powerful alternative to Happy but in fact its main usage is to describe natural language grammars instead of programming languages. The language itself will look familiar for most Haskell or ML users. It is a dependently typed functional language based on Per Martin-Löf's type theory.

An important objective in the language development was to make it possible to develop modular grammars. The language provides modular system inspired from ML but adapted to the specific requirements in GF. The modules system was exploited to a large extent in the Resource Libraries project. The library provides large linguistically motivated grammars for a number of languages. When the languages are closely related the common parts in the grammar could be shared using the modules system. Currently there are complete grammars for Amharic, Bulgarian, Catalan, Danish, Dutch, English, Finnish, French, German, Interlingua, Italian, Norwegian, Russian, Spanish, Swedish and Urdu. There are also incomplete grammars for Arabic, Latin, Thai, Turkish, and Hindi. On top of these grammars a user with limited linguistic background can build application grammars for a particular domain.

We are planning the release of GF 3.2 before the end of this year. The latest development around GF is mostly driven by the new research project MOLTO (http://www.molto-project.eu/) which focuses on tools for multilingual online translation. This are some of the features that can be expected in the new release:

- We improve our web front-end so the users (translators, content authors, etc.) will work in a more comfortable environment. There are some demos on the GF home page.
- There is work in progress on a new editor which combines free text authoring with structural editing.
- Now there is a web based browser which the grammarians can use to explore the content of the grammars. The functionality is similar to what you would expect from tools like Haddock for Haskell.
- The support for dependent types is becoming stable. All abstract syntax trees are type checked before they

are used. We use the same type checking algorithm as in Agda and we support the Agda style of implicit arguments. If the abstract syntax in some grammar uses dependent types then the parser checks whether the parsed sentence is semantically consistent. When GF is used for parsing formal languages like C/C++ then the concrete syntax of the grammar describes the syntax of the language while the dependent types in the abstract syntax can be used to specify which programs are well-typed.

◦ We also support random and exhaustive generation of lambda terms of a given type. Since the type system supports dependent types, the generation is actually equivalent to proving a theorem in first-order logic. In fact, we build our own in-house theorem prover.

◦ Previously the parser in GF either produced some result or just failed. Now it is much more friendly and could detect where exactly is the problem. If there is a syntax error then the token position is reported. If the parsing is successful but none of the possible parse trees is semantically consistent then the inconsistent phrase is located and a detailed error message is reported. In some cases semantic inconsistencies can be detected even before the sentence is complete. In formal languages, this corresponds to type checking of incomplete programs.

◦ There is an alternative implementation of the GF interpreter in Java which makes it possible to run applications on platforms where Haskell is not well supported. For instance we developed a user interface which works on Android phones.

Further reading

http://www.grammaticalframework.org/

9.8 Others

9.8.1 xmonad

Report by:	Gwern Branwen
Status:	active development

XMonad is a tiling window manager for X. Windows are arranged automatically to tile the screen without gaps or overlap, maximizing screen use. Window manager features are accessible from the keyboard; a mouse is optional. XMonad is written, configured, and extensible in Haskell. Custom layout algorithms, key bindings, and other extensions may be written by the user in config files. Layouts are applied dynamically, and different layouts may be used on each workspace. Xinerama is fully supported, allowing windows to be tiled on several physical screens.

Development since the last report has continued apace, with versions 0.8, 0.8.1, 0.9 and 0.9.1 released, with simultaneous releases of the XMonadContrib library of customizations and extensions, which has now grown to no less than 205 modules encompassing a dizzying array of features.

Details of changes between releases can be found in the release notes:

◦ http://haskell.org/haskellwiki/Xmonad/Notable_changes_since_0.7
◦ http://haskell.org/haskellwiki/Xmonad/Notable_changes_since_0.8
◦ http://haskell.org/haskellwiki/Xmonad/Notable_changes_since_0.9
◦ XMonad.Config.PlainConfig allows writing configs in a more 'normal' style, and not raw Haskell
◦ Supports using local modules in xmonad.hs; for example: to use definitions from ~/.xmonad/lib/XMonad/Stack/MyAdditions.hs
◦ xmonad –restart CLI option
◦ xmonad –replace CLI option
◦ XMonad.Prompt now has customizable keymaps
◦ Actions.GridSelect - a GUI menu for selecting windows or workspaces
◦ Actions.OnScreen
◦ Extensions now can have state
◦ Actions.SpawnOn - uses state to spawn applications on the workspace the user was originally on, and not where the user happens to be
◦ Markdown manpages and not man/troff
◦ XMonad.Layout.ImageButtonDecoration & XMonad.Util.Image
◦ XMonad.Layout.Groups
◦ XMonad.Layout.ZoomRow
◦ XMonad.Layout.Renamed
◦ XMonad.Layout.Drawer
◦ XMonad.Hooks.ScreenCorners
◦ XMonad.Actions.DynamicWorkspaceOrder
◦ XMonad.Actions.WorkspaceNames
◦ XMonad.Actions.DynamicWorkspaceGroups

Binary packages of XMonad and XMonadContrib are available for all major Linux distributions.

Further reading

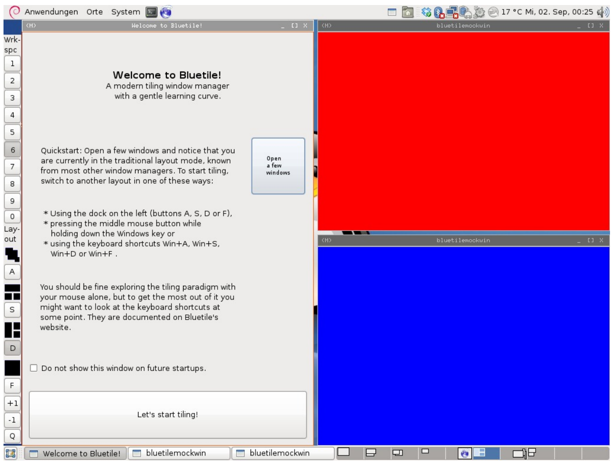
◦ Homepage: http://xmonad.org/
◦ Darcs source:
`darcs get` http://code.haskell.org/xmonad
◦ IRC channel: `#xmonad @@ irc.freenode.org`
◦ Mailing list: ⟨xmonad@haskell.org⟩

9.8.2 Bluetile

Report by:	Jan Vornberger
Status:	active development

Bluetile is a tiling window manager for X based on xmonad ((→ 9.8.1)). Windows are arranged to use the entire screen without overlapping. Bluetile's focus lies on making the tiling paradigm easily accessible to users coming from traditional window managers by drawing on known conventions and providing both mouse and keyboard access for all features. It also tries to be usable "out of the box", requiring minimal to no configuration in most cases.

- Hybrid approach: Stacking window layout & tiling layouts available
- Maximizing & minimizing windows in all layouts
- All features accessible from mouse, as well as keyboard
- Good multihead support
- Proper handling of fullscreen applications
- Designed to integrate with the GNOME desktop environment



Welcome to Bluetile!
A modern tiling window manager
with a gentle learning curve.

Quickstart: Open a few windows and notice that you are currently in the traditional layout mode, known from most other window managers. To start tiling, switch to another layout in one of these ways:

* Using the dock on the left (buttons A, S, D or F),
* pressing the middle mouse button while holding down the Windows key or
* using the keyboard shortcuts Win+A, Win+S, Win+D or Win+F .

You should be fine exploring the tiling paradigm with your mouse alone, but to get the most out of it you might want to look at the keyboard shortcuts at some point. They are documented on Bluetile's website.

☐ Do not show this window on future startups.

Let's start tiling!

Further reading

http://www.bluetile.org/

9.8.3 Biohaskell

Report by:	Ketil Malde
Participants:	Christian Höner zu Siederdissen

Bioinformatics in Haskell is a steadily growing field, and the relevant section on Hackage now sports several libraries and applications. The original *bioinformatics library* supports working with nucleotide and protein sequences and associated data. It supports a variety of file and alignment formats, and provides basic functions for working with sequences.

This library has been used in a number of applications, the latest are *flowsim*, a simulator for 454-style sequences and *a50*, a tool for assessing the quality of genome assemblies.

Lately, several libraries dealing with RNA structure and analysis have been added. These support working with thermodynamics parameters (most widely known as the Turner set of parameters) and RNA family models as used by the Infernal Covariance Model algorithms.

Algorithms depending on those libraries are a re-implementation of *RNAfold* of the *ViennaRNA* package and a novel tool, called *CMCompare*, which provides a first algorithm to compare *Infernal* RNA family models. Re-implemented programs serve as a testbed for efficient arrays for Haskell (mainly the vector library). Several new algorithms are currently in development.

Further reading

- http://blog.malde.org/
- http://www.tbi.univie.ac.at/~choener/Haskell/

9.8.4 IgorII

Report by:	Martin Hofmann
Participants:	Emanuel Kitzelmann, Ute Schmid
Status:	experimental, active development

IGORII is a new method and an implemented prototype for constructing recursive functional programs from a few non-recursive, possibly non-ground, example equations describing a subset of the input/output behavior of a target function to be implemented.

For a simple target function like `reverse` the sole input would be the following, the k smallest w.r.t. the input data type, examples:

```
reverse []      = []
reverse [a]     = [a]
reverse [a,b]   = [b,a]
reverse [a,b,c] = [c,b,a]
```

The result, shown below, computed by IGORII is a recursive definition of `reverse`, where the subfunctions `last` and `init` have been automatically invented by the program.

```
reverse []     = []
reverse (x:xs) = (last (x:xs)):(reverse (init (x:xs))

last [x]       = x
last (x:y:ys)  = last (y:ys)
init [x]       = []
init (x:y:ys)  = x:(init (y:ys))
```

IGORII has been extended to use catamorphisms on lists as higher-order templates. After enabling the higher-order mode, given the previous examples of `reverse`, the system outputs the following solution:

```
reverse xs    = foldr snoc [] xs

snoc x xs     = foldr cons [x] xs
cons x (y:ys) = x:(y:ys)
```

Recently, the use of list-catamorphism has been generalized to arbitrary inductive data types. Based on the Pointless Haskell library IgorII uses its generic implementation of recursion patterns to solve recursive problems as e.g. *mirroring binary trees*, *computing the power set*, or to find a recursive solution for the *towers of hanoi*, to mention just a few.

Features

- termination by construction
- handling arbitrary user-defined data types
- utilization of arbitrary background knowledge
- automatic invention of auxiliary functions as subprograms
- learning complex calling relationships (tree- and nested recursion)
- allowing for variables in the example equations
- simultaneous induction of mutually recursive target functions
- using catamorphisms on arbitrary inductive data types as higher-order templates or generic recursion schemes

Current Status and Future Plans

The original version of IgorII is implemented in the reflective rewriting based programming and specification language Maude. However, a Haskell implementation of the algorithm is the current research prototype. Both can be obtained from the project page.

A tool demo and a research paper about the use of catamorphisms as higher-order templates were presented at PEPM 2010.

For the future, we plan to extend the system to use other type morphisms as generic recursion schemes. Also it would be worth investigating to which extent knowledge about types, e.g. universal properties, can be used for the synthesis process, e.g. to guide the search or resolve ambiguities.

Further reading

- http://www.cogsys.wiai.uni-bamberg.de/effalip/
- http://www.inductive-programming.org/

9.8.5 arbtt

Report by:	Joachim Breitner
Status:	working

The program arbtt, the automatic rule-based time tracker, allows you to investigate how you spend your time, without having to manually specify what you are doing. arbtt records what windows are open and active, and provides you with a powerful rule-based language to afterwards categorize your work. And it comes with documentation!

Further reading

- http://www.joachim-breitner.de/projects#arbtt
- http://www.joachim-breitner.de/blog/archives/336-The-Automatic-Rule-Based-Time-Tracker.html
- http://darcs.nomeata.de/arbtt/doc/users_guide/

9.8.6 cltw (Twitter API Command-Line Utility)

Report by:	Dino Morelli
Status:	experimental, actively developed

This is a tool for performing some Twitter API functions from the command-line. So far supporting three calls: statuses/followers, statuses/friends, statuses/update.

cltw is available from Hackage, the darcs repository below, and also in binary form for Arch Linux through the AUR.

Further reading

- Project page: http://ui3.info/d/proj/cltw.html
- Source repository: `darcs get` http://ui3.info/darcs/cltw

10 Commercial Users

10.1 Well-Typed LLP

Report by:	Ian Lynagh
Participants:	Duncan Coutts, Andres Löh, Dmitry Astapov

Well-Typed is a Haskell services company. We provide commercial support for Haskell as a development platform, including consulting services, training, and bespoke software development. For more information, please take a look at our website or drop us an e-mail at ⟨info@well-typed.com⟩.

Business continues to go well, and the Well-Typed team has therefore doubled in size since the last HCAR edition. Andres Löh has become the third partner, and Dmitry Astapov has joined as a subcontractor.

The new manpower is in part for the "Parallel Haskell Project", which has now begun. This is a 2-year project, funded by Microsoft Research, to push the real-world adoption and practical development of parallel Haskell with GHC. We are working with four companies that want to make use of parallel Haskell. Our job is to help them succeed, including fixing any problems with the tools that they might run into.

The second round of the Industrial Haskell Group's Collaborative Development Scheme has also just begun. Expect to see some updates on what the IHG is getting up to on our blog, and a summary in the next HCAR.

Further reading

◦ http://www.well-typed.com/
◦ Blog: http://blog.well-typed.com/

10.2 Bluespec Tools for Design of Complex Chips and Hardware Accelerators

Report by:	Rishiyur Nikhil
Status:	commercial product

Bluespec, Inc. provides a language, BSV, which is being used for all aspects of ASIC and FPGA system design — specification, synthesis, modeling, and verification. All hardware behavior is expressed using *rewrite rules* (Guarded Atomic Actions). BSV borrows many ideas from Haskell — algebraic types, polymorphism, type classes (overloading), and higher-order functions. Strong static checking extends into correct expression of multiple clock domains, and to gated clocks for power management. Unlike HW design with C, which can only be used for "loop-and-array" computations, BSV is universal, accommodating the diverse range of blocks found in modern SoCs, from algorithmic "datapath" blocks to complex control blocks such as processors, DMAs, interconnects, and caches.

Bluespec's core tool synthesizes (compiles) BSV into high-quality RTL (Verilog), which can be further synthesized into netlists for ASICs and FPGAs using other commercial tools. Automatic synthesis from atomic transactions enables design-by-refinement, where an initial executable approximate design is systematically transformed into a quality implementation by successively adding functionality and architectural detail. The core tool is implemented in Haskell (well over 100K lines).

In addition to the core synthesis tool, Bluespec provides a fast simulation tool for BSV, and extensive libraries and infrastructure to make it easy to build FPGA-based accelerators for computationally intensive software, including for the Xilinx XUP board popular in universities.

These industrial strength tools have enabled some large designs (over a million gates) and significant architecture research projects in academia and industry, because complex architectural models can now be coded with the same convenience of expression as SW languages, but with the execution speed of FPGAs.

Status and availability

BSV tools, available since 2004, are in use by several major semiconductor and electronic equipment companies, and universities. The tools are free for academic teaching and research.

Further reading

◦ R.S.Nikhil, *Bluespec, a General-Purpose Approach to High-Level Synthesis Based on Parallel Atomic Transactions*, in *High Level Synthesis: from Algorithm to Digital Circuit, Philippe Coussy and Adam Morawiec (editors)*, Springer, 2008, pp. 129-146.
◦ Small illustrative examples: http://sites.google.com/a/bluespec.com/learning-bluespec/Home/Small-Examples
◦ MIT courseware, "Complex Digital Systems": http://csg.csail.mit.edu/6.375
◦ A fun example with many functional-programming features — BluDACu, a parameterized Bluespec hardware implementation of Sudoku: http://www.bluespec.com/products/BluDACu.htm

10.3 Industrial Haskell Group

Report by:	Ian Lynagh
Participants:	Duncan Coutts, Andres Löh, Dmitry Astapov

The Industrial Haskell Group (IHG) is an organization to support the needs of commercial users of Haskell. It was formed in early 2009, and in the first 6 month collaborative development scheme funded work on dynamic libraries, more flexible Integer library support for GHC, and Cabal development work.

More recently, on behalf of the first of our university members, we have diagnosed an issue with using GLUT from within GHCi on Mac OS X. This issue will be resolved in the upcoming GHC 7.0.1 release.

Meanwhile, the second iteration of the collaborative development scheme is getting underway; details about the tasks undertaken will be appearing on the Well-Typed (→ 10.1) blog, and you can expect a summary in the next HCAR edition.

We expect to run the collaborative development scheme continuously, so if you are interested in joining as a full member, please get in touch. Details of this, as well as the associate and academic membership options, are on the website.

If you are interested in joining the IHG, or if you just have any comments, please drop us an e-mail at ⟨info@industry.haskell.org⟩.

Further reading

http://industry.haskell.org/

10.4 factis research GmbH

Report by:	Stefan Wehr
Participants:	David Leuschner, Harald Fischer
Status:	beta, active development

factis research, located in Freiburg, Germany, develops reliable and user-friendly mobile solutions. Our client software runs under J2ME, Symbian, iPhone OS, Android, and Blackberry. The server components are implemented in Python and Haskell.

We are actively using Haskell for a number of projects, most of which are released under an open-source license:

- Server backends for our mobile software solutions.
- *DPM* (→ 6.5.4), a patch manager for darcs.
- *HTF* (→ 6.3.1), a test framework.
- *fos* (http://openfactis.org/fos/), a customer relationship management tool. Originally, fos was written as a Haskell GTK application, but we are currently rewriting it as a web application.
- *ntee* (http://openfactis.org/ntee), an adaptation of the Unix tool `tee` to network streams.

Further reading

http://www.factisresearch.com/

10.5 Tsuru Capital

Report by:	Simon Cranshaw

Tsuru Capital is engaged in high-frequency market-making on options markets. Tsuru is a private company, and trades with its own capital. Tsuru Capital currently runs arbitrage based liquidity provision strategies on the Kospi 200 index and plans to expand to Nikkei 225 index, and other electronic markets, over the next year.

The trading software has been developed entirely in Haskell, and is one of the few systems in the world written completely in a functional language.

Further reading

http://tsurucapital.com/en/

10.6 Oblomov Systems

Report by:	Martijn Schrage

Oblomov Systems

Oblomov Systems is a one-person software company based in Utrecht, The Netherlands. Founded in 2009 for the Proxima 2.0 project (→ 9.2.6), Oblomov has since then been working on a number of Haskell-related projects. The main focus lies on web-applications and (web-based) editors. Haskell has turned out to be extremely useful for implementing web servers that communicate with JavaScript clients or iPhone apps.

Awaiting the acceptance of Haskell by the world at large, Oblomov Systems also offers software solutions in Java, Objective C, and C#, as well as on the iPhone.

Further reading

http://www.oblomov.com

11 Research and User Groups

11.1 Artificial Intelligence and Software Technology at Goethe-University Frankfurt

Report by:	David Sabel
Participants:	Conrad Rau, Manfred Schmidt-Schauß

One of our research topics focuses on programming language semantics, especially on contextual equivalence which is usually based on the operational semantics of the language. We explored several call-by-need lambda calculi. Deterministic call-by-need lambda calculi with letrec provide a semantics for the core language of Haskell. In the setting of such an extended calculus we proved correctness of strictness analysis using abstract reduction. Furthermore, we proved equivalence of the call-by-name and call-by-need semantics of an extended lambda calculus with `letrec`, `case`, and constructors.

Recently, we extended our investigations to parametric polymorphism and showed correctness of type dependent program transformations. Most recently, in collaboration with *Elena Machkasova* we have shown that the call-by-need lambda calculus with letrec is isomorphic to the lazy lambda calculus and that bisimilarity coincides with contextual equivalence in the call-by-need lambda calculus with letrec.

We also explored several nondeterministic extensions of call-by-need lambda calculi and their applications. We analyzed a model for a lazy functional language with direct-call I/O providing a semantics for `unsafePerformIO` in Haskell. We investigated a call-by-need lambda calculus extended with McCarthy's `amb` and an abstract machine for lazy evaluation of concurrent computations. We have shown that mutual similarity is a sound proof method w.r.t. contextual equivalence in a class of untyped higher-order nondeterministic call-by-need lambda calculi. A recent result is that the situation changes if recursive let is added to such a calculus, i.e., for calculi with `letrec` and nondeterminism all usual definitions of similarity are unsound w.r.t. contextual equivalence.

In a recently started research project we try to automatize correctness proofs of program transformations. A main step for this goal is the computation of overlappings between reductions of the operational semantics and transformations steps. This computation requires the combination of several unification algorithms. We implemented a prototype of this combined algorithm in Haskell.

As a further research topic we analyzed the expressivity of concurrency primitives in various functional languages. In collaboration with *Jan Schwinghammer* and *Joachim Niehren*, we showed how to encode Haskell's MVars into a lambda calculus with storage and futures which is an idealized core language of Alice ML. We proved correctness of the encoding using operational semantics and the notions of adequacy and full-abstractness of translations.

Further reading

http://www.ki.informatik.uni-frankfurt.de/research/HCAR.html

11.2 Functional Programming at the University of Kent

Report by:	Olaf Chitil

The Functional Programming group at Kent is a subgroup of the Programming Languages and Systems Group of the School of Computing. We are a group of staff and students with shared interests in functional programming. While our work is not limited to Haskell — in particular our interest in Erlang has been growing — Haskell provides a major focus and common language for teaching and research.

Our members pursue a variety of Haskell-related projects, some of which are reported in other sections of this report. Simon Thompson is close to completing a third edition of his Haskell text book, due out in spring 2011. The Haskell Refactorer Hare (→6.1.5) has recently been cabal-ised, and now features clone detection and elimination facilities. Thomas Schilling is developing ideas for improving type error messages for GHC and on trace-based dynamic optimisations for Haskell programs. Olaf Chitil is working on better lazy assertions for Haskell.

Further reading

- PLAS group: http://www.cs.kent.ac.uk/research/groups/plas/
- Refactoring Functional Programs: http://www.cs.kent.ac.uk/research/groups/plas/hare.html
- Tracing and debugging with Hat: http://www.haskell.org/hat
- Heat: http://www.cs.kent.ac.uk/projects/heat/
- Scion: http://code.google.com/p/scion-lib/

11.3 Formal Methods at DFKI and University Bremen

Report by:	Christian Maeder
Participants:	Mihai Codescu, Dominik Dietrich, Dominik Lücke, Christoph Lüth, Till Mossakowski, Lutz Schröder, Ewaryst Schulz
Status:	active development

The activities of our group center on formal methods, covering a variety of formal languages and also translations and heterogeneous combinations of these.

We are using the Glasgow Haskell Compiler and many of its extensions to develop the Heterogeneous tool set (Hets). Hets consists of parsers, static analyzers, and proof tools for languages from the CASL family, such as the Common Algebraic Specification Language (CASL) itself, HasCASL, CoCASL, CspCASL, and ModalCASL. Other languages supported include Haskell (via Programatica), QBF, Maude, VSE, TPTP, OWL, Common Logic, and LF type theory. The Hets implementation is also based on some old Haskell sources such as bindings to uDrawGraph (formerly Davinci) and Tcl/TK that we maintain. A RESTful interface to Hets is under development.

HasCASL is a general-purpose higher-order language which is in particular suited for the specification and development of functional programs; Hets also contains a translation from an executable HasCASL subset to Haskell. There is a prototypical translation of a subset of Haskell to Isabelle/HOL and HOLCF.

The Coalgebraic Logic Satisfiability Solver CoLoSS is being implemented jointly at DFKI Bremen and at the Department of Computing, Imperial College London. The tool is generic over representations of the syntax and semantics of certain modal logics; it uses the Haskell class mechanism, including multi-parameter type classes with functional dependencies, extensively to handle the generic aspects.

Further reading

- Group activities overview:
  http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/
- CASL specification language:
  http://www.cofi.info
- Heterogeneous tool set:
  http://www.dfki.de/sks/hets
  http://www.informatik.uni-bremen.de/htk/
  http://www.informatik.uni-bremen.de/uDrawGraph/
- The Coalgebraic Logic Satisfiability Solver CoLoSS:
  http://www.informatik.uni-bremen.de/~lschrode/projects/GenMod
  http://www.doc.ic.ac.uk/~dirk/COLOSS/

11.4 Haskell at Universiteit Gent, Belgium

Report by:	Tom Schrijvers

Haskell is one of the main research topics of the new Programming Languages Group at the Department of Applied Mathematics and Computer Science at the University of Ghent, Belgium.

Haskell-related projects of the group members and collaborators are:

- *The Monad Zipper*: Limitations of monad stacks get in the way of developing highly modular programs with effects. This pearl demonstrates that Functional Programming's abstraction tools are up to the challenge. Of course, abstraction must be followed by clever instantiation: Huet's zipper for the monad stack makes components jump through unanticipated hoops. This is joint work with Bruno Oliveira, available together with Mauro Jaskelioff's monad transformer library in the Monatron package on Hackage.

- *EffectiveAdvice:* EffectiveAdvice is a disciplined model of (AOP-style) advice, inspired by Aldrich's Open Modules, that has full support for effects in both base components and advice. EffectiveAdvice is implemented as a Haskell library. Advice is modeled by mixin inheritance and effects are modeled by monads. Interference patterns previously identified in the literature are expressed as combinators. Equivalence of advice, as well as base components, can be checked by equational reasoning. Parametricity, together with the combinators, is used to prove two harmless advice theorems. The result is an effective model of advice that supports effects in both advice and base components, and allows these effects to be separated with strong non-interference guarantees, or merged as needed. This is joint work with Bruno Oliveira and William Cook.

- *Type Checking:* Recent results are on type inference for GADTs, type invariants, and type checking for type families. Ongoing work concerns the simplification of type checking for Haskell extensive type system, and adding new extensions. This is joint work with Martin Sulzmann, Simon Peyton Jones, Manuel Chakravarty, Dimitrios Vytiniotis, Stefan Monnier, Louis-Julien Guillemette, and Dominic Orchard.

- *The Monadic Constraint Programming Framework:* The main article on the MCP framework by Tom Schrijvers, Peter Stuckey and Phil Wadler has appeared in the Journal of Functional Programming. It explains how the framework captures the generic aspects of Constraint Programming in Haskell. Of particular interest is the solver-independent framework for compositional search strategies.

Currently we are extending the framework to act as a finite domain modeling language for both the problem description and the search component. The model in Haskell serves as a high-level front-end for a state-of-the-art Constraint Programming system such as Gecode (C++). Models can be compiled to C++ code, can be solved by calling Gecode from Haskell at runtime, or can be solved purely in Haskell itself. This is joint work with Pieter Wuille and Guido Tack.

We are also involved in the organization of the Ghent Functional Programming Group (→ 11.10).

Further reading

- http://users.ugent.be/~tschrijv/Haskell/
- http://hackage.haskell.org/package/Monatron
- http://hackage.haskell.org/package/monadiccp

11.5 fp-syd: Functional Programming in Sydney, Australia

Report by:	Erik de Castro Lopo
Participants:	Ben Lippmeier, Shane Stephens, and others

We are a seminar and social group for people in Sydney, Australia, interested in Functional Programming and related fields. We have 10 meetings per year (Feb–Nov) and meet on the third Thursday of each month. We regularly get 20–30 attendees, with a 70/30 industry/research split. Talks this year have included material on Arrows, Scala's Actors, Pattern Calculus, a couple of different Haskell libraries, and the ATS programming language. We usually have about 90 mins of talks, starting at 6:30pm, then go for drinks afterwards. All welcome.

Further reading

http://groups.google.com/group/fp-syd

11.6 Functional Programming at Chalmers

Report by:	Jean-Philippe Bernardy

Functional Programming is an important component of the Department of Computer Science and Engineering at Chalmers. In particular, Haskell has a very important place, as it is used as the vehicle for teaching and numerous projects. Besides functional programming, language technology, and in particular domain specific languages is a common aspect in our projects.

Property-based testing QuickCheck is the basis for a European Union project on Property Based Testing (www.protest-project.eu). We are applying the QuickCheck approach to Erlang software, together with Ericsson, Quviq, and others. Much recent work has focused on PULSE, the ProTest User-Level Scheduler for Erlang, which has been used to find race conditions in industrial software — see our ICFP 2009 paper for details. A new tool, QuickSpec, generates algebraic specifications for an API automatically, in the form of equations verified by random testing. We have published about it at TAP 2010; an earlier paper can be found here: http://www.cse.chalmers.se/~nicsma/quickspec.pdf. Lastly, we have devised a technique to speed up testing of polymorphic properties: http://publications.lib.chalmers.se/cpl/record/index.xsql?pubid=99387.

Natural language technology Grammatical Framework (→ 9.7.3) is a declarative language for describing natural language grammars. It is useful in various applications ranging from natural language generation, parsing and translation to software localization. The framework provides a library of large coverage grammars for currently fifteen languages from which the developers could derive smaller grammars specific for the semantics of a particular application.

Parser generator and template-haskell BNFC-meta is a parser generator. Like the BNF Converter, it generates a compiler front end in Haskell. Two things separate BNFC-meta from BNFC and other parser generators:

- BNFC-meta is not a program but a library (the parser description is embedded in a quasi-quote).
- BNFC-meta automatically provides quasi-quotes for the specified language. This includes a powerful and flexible facility for antiquotation.

More info: http://hackage.haskell.org/package/BNFC-meta.

Generic Programming Starting with Polytypic Programming in 1995 there is a long history of generic programming research at Chalmers. Recent developments include fundamental work on parametricity & dependent types (ICFP 2010), a survey paper "Generic programming with C++ concepts and Haskell type classes" (JFP 2010) and two new PhD students. Patrik Jansson leads a work-package on DSLs within the EU project "Global Systems Dynamics and Policy" (http://www.gsdp.eu/, started Oct. 2010). If you want to apply DSLs, Haskell, and Agda to help modelling global sustainability challenges, please get in touch!

Language-based security SecLib is a light-weight library to provide security policies for Haskell programs. The library provides means to preserve confidentiality of data (i.e., secret information is not leaked) as well as the ability to express intended releases of information known as declassification. Besides confidentiality policies, the library also supports another important aspect of security: integrity of data. SecLib provides an attractive, intuitive, and simple setting to explore the security policies needed by real programs.

Type theory Type theory is strongly connected to functional programming research. Many dependently-typed programming languages and type-based proof assistants have been developed at Chalmers. The Agda system (→ 4.1) is the latest in this line, and is of particular interest to Haskell programmers. We encourage you to experiment with programs and proofs in Agda as a "dependently typed Haskell".

DSP programming Feldspar is a domain-specific language for digital signal processing (DSP), developed in co-operation by Ericsson, Chalmers FP group and Eötvös Loránd (ELTE) University in Budapest. The motivating application is telecom processing, but the language is intended to be more general. As a first stage, we have focused on the data-intensive numeric algorithms which are at the core of any DSP application. More recently, we have started to work on extending the language to deal with more system-level aspects. The data processing language is purely functional and highly inspired by Haskell. Currently the language is implemented as an embedded language in Haskell.

The implementation is available from Hackage: http://hackage.haskell.org/package/feldspar-language. There is also a code generator, developed at ELTE University: http://hackage.haskell.org/package/feldspar-compiler.

See also the official project page: http://feldspar.inf.elte.hu.

Hardware design/verification The functional programming group has developed three different hardware description languages — Lava, Wired, and Chalk (chronological order) — implemented in Haskell. Each language targets a different abstraction level. The basic idea behind all three is to model circuits as functions from inputs to outputs. This allows structural hardware description in standard functional programming style.

Chalk is a new language for architecture design. Once you have defined a Chalk circuit, you can simulate it, or explore it further using non-standard interpretations. This is particularly useful if you want to perform high-level power and performance analysis early on in the design process.

More info: http://www.cse.chalmers.se/~wouter/Publications/DCC2010.pdf.

In **Lava**, circuits are described at the gate level (with some RTL support). The version developed at Chalmers has a particular aim to support formal verification in a convenient way.

Wired is an extension to Lava, targeting (not exclusively) semi-custom VLSI design. A particular aim of Wired is to give the designer more control over on-chip wires' effects on performance. Some features of Wired are:

- Initial description can be purely functional (a la Lava).
- Incremental specification of physical aspects.
- Accurate, wire-aware timing/power analysis within the system.
- Support for an academic 45nm cell library.

Wired is not actively developed at the moment, but the system has recently been used to explore the layout of multipliers (Kasyab P. Subramaniyan, Emil Axelsson, Mary Sheeran and Per Larsson-Edefors. Layout Exploration of Geometrically Accurate Arithmetic Circuits. *Proceedings of IEEE International Conference of Electronics, Circuits and Systems.* 2009).

Home page: http://www.cse.chalmers.se/~emax/wired/.

Automated reasoning Equinox is an automated theorem prover for pure first-order logic with equality. Equinox actually implements a hierarchy of logics, realized as a stack of theorem provers that use abstraction refinement to talk with each other. In the bottom sits an efficient SAT solver. Paradox is a finite-domain model finder for pure first-order logic with equality. Paradox is a MACE-style model finder, which means that it translates a first-order problem into a sequence of SAT problems, which are solved by a SAT solver. Infinox is an automated tool for analyzing first-order logic problems, aimed at showing finite unsatisfiability, i.e., the absence of models with finite domains. All three tools are developed in Haskell.

Teaching Haskell is present in the curriculum as early as the first year of the Bachelors program. We have three courses solely dedicated to functional programming (of which two are Masters-level courses), but we also provide courses which use Haskell for teaching other aspects of computer science, such as programming languages, compiler construction, hardware description and verification, data structures and programming paradigms.

11.7 Dutch Haskell User Group

Report by:	Tom Lokhorst

The Dutch Haskell User Group is a diverse group of people interested in Haskell and functional programming.

Since the inception of our user group in April of 2009, we have had monthly meetings and an afternoon symposium. Our meetings alternate between pure socializing and evenings that include talks by members.

We have a wide range of international members; people using functional programming in academia, as a hobby, or for commercial purposes.

Anyone is welcome to join, from beginners to advanced users. Do join us!

Further reading

http://dutchhug.nl/

11.8 San Simón Haskell Community

Report by:	Carlos Gomez

The San Simón Haskell Community from San Simón University Cochabamba-Bolivia, is an informal Spanish group that search to learn, share information, knowledge and experience related to the functional paradigm.

Since more than a year, we are trying to expand our community all across Latin American Haskell programmers, and in order to do that, we created a web page (http://comunidadhaskell.org) that serves us as a medium of communication and work environment. All Haskell programmers are welcome to contribute to this site.

Our main activity is the development of projects, and related to that we have information links, a wiki, a blog, some materials and lately we have a section for challenges related to Haskell (http://challenges.comunidadhaskell.org). We started an event for every year in which we present the projects of the last year. On 15th April 2010, we celebrated our 1st Open House Haskell Community in which we presented our projects.

You can also meet us on Facebook, this community is open to all Haskell programmers and specially to Spanish Haskell programmers.

Further reading

http://comunidadhaskell.org

11.9 Functional Programming at KU

Report by:	Andy Gill
Status:	ongoing

Functional Programming remains active at KU and the Computer Systems Design Laboratory in ITTC. The System Level Design Group (lead by Perry Alexander) and the Functional Programming Group (lead by Andy Gill) together form the core functional programming initiative at KU. Apart from Kansas Lava (→ 9.5.3) and ChalkBoard (→ 8.7.4), there are many other FP and Haskell related things going on.

◦ We are developing a Haskell version of HOL. Traditionally, members of the higher-order logic theorem (HOL) proving family have been implemented in the Standard ML programming language or one of its derivatives. HaskHOL aims to break with tradition by implementing a lightweight HOL theorem prover library as a Haskell hosted domain specific language. Based on the HOL Light logical system, HaskHOL aims to provide the ability for Haskell users to reason about their code directly without having to transform it or otherwise export it to an external tool. For details talk to Evan Austin.

◦ We are actively working on enabling *Type-Directed Specification Refinement in Rosetta*. Rosetta is a specification language that focuses on the interaction between different domains, such as state-based and signal-based domains. With dependent types, first-class types, and reflection, there are many areas where a traditional all-or-nothing typing analysis would be impractical — especially when considering that specifications are likely written at first in a high-level, incomplete fashion. This project uses InterpreterLib (http://haskell.org/communities/11-2008/html/report.html#sect5.5.6) and various Rosetta analysis tools to define a typing analysis that attempts to extract typing information, constraints, and errors to present to the user, in order to guide the specification refinement process. It is in the early stages of development, but may eventually link up with HaskHOL to discharge some TCC's. For details talk to Mark Snyder.

- We are developing a library in Haskell for processing Rosetta specifications. A current focus is the modularity and re-use of distinct processing elements, such as type-checking, partial evaluation, and reasoning assistants. Mutually defined elements that are more convenient to consider as distinct interact via a reactive monadic computation, so the two elements' code can be managed as separate packages. Also, our principal specification representation use functors and type-level fixed points to achieve extensibility and generic programming. The goal of the library is to provide to a tight and graduated interface to the basic processing elements, so that the users may incorporate the most appropriate basic elements when implementing their own, more domain-specific Rosetta processors. For details talk to Nick Frisby.

- We are working with other functional programming groups (University of Iowa, St. Andrews, Heriot-Watt, Halmstad University, and of course Chalmers) to share our common experiences with using FPGA boards, and generating VHDL. So far, we have chosen and purchased common Xilinx boards, and have a design for a so called "λ-bridge" between our UNIX invocation infrastructures and our FPGA boards. The idea is we can share experiences, for the sake of being able to spend more time working on FP issues, and bringing FP ideas to hardware related problems.

Further reading

- The Functional Programming Group: http://www.ittc.ku.edu/csdl/fpg
- CSDL website: https://wiki.ittc.ku.edu/csdl/Main_Page

11.10 Ghent Functional Programming Group

Report by:	Jeroen Janssen
Participants:	Bart Coppens, Jasper Van der Jeugt, Tom Schrijvers, Andy Georges, Kenneth Hoste
Status:	active

GHENT FPG

The Ghent Functional Programming Group is a new user group aiming to bring together programmers, academics, and others interested in functional programming located in the area of Ghent, Belgium. Our goal is to have regular meetings with talks on functional programming, organize functional programming related events such as hackathons, and to promote functional programming in Ghent by giving after-hours tutorials.

The first two meetings were reported on in the last edition of HCAR. The third meeting was on June 29, 2010. The program was as follows:

1. Pierre Carbonnelle — Declarative programming for Business Logic: a new open-source project

2. Tom Schrijvers — Monadic Constraint Programming

3. Wouter Kampmann, Lieven Lemiengre — Lightning Talk on Scala

The fourth meeting was on October 7, 2010. The program was as follows:

1. Stijn Timbermont — Mapping Interpreters onto Runtime Support

2. Tom Schrijvers — Dictionaries: Eager or Lazy Type Class Witnesses?

3. Dominique Devriese — Grammar Combinators - A new model for shallow parser DSLs

On November 5–7, 2010 we hosted BelHac, the first Belgian Haskell Hackathon, kindly sponsored by Incubaid (http://www.incubaid.com), Well-Typed (→ 10.1), and O'Reilly (http://www.oreilly.com).

On Friday, we started with a great introduction to Haskell by Miran Lipovača, the author of http://learnyouahaskell.com. A lot of students from Ghent University showed up for this tutorial — it is good to see that there is an interest in Haskell among students. In parallel, there was some hacking, since some people already knew Haskell.

After that, we moved to another (more fancy) building to listen to talks by Duncan Coutts, Romain Slootmaekers and Don Stewart. The talks all focused around "Haskell in the Industry", and, quoting Don Stewart, "felt like startup school for Haskellers". Perhaps we will see some new Haskell startups soon?

The talks were followed by a small reception, and after that, some obligatory Belgian beers in nearby pubs.

On Saturday and Sunday, the focus was on hacking. We had a small contest: since we had some shiny new Real World Haskell books to give away, we made a list of people who uploaded a package to Hackage during the Hackathon. On Sunday, three lucky winners were chosen.

We think this Hackathon was a success. Patches were added to cabal, to the new Hackage, and, of course, a lot of packages were released. We hope to see everyone again at the next Hackathon!

If you want more information on GhentFPG you can follow us on twitter (@ghentfpg), via Google Groups (http://groups.google.com/group/ghent-fpg), or by visiting us at irc.freenode.net in channel #ghentfpg.

Further reading

http://groups.google.com/group/ghent-fpg