Haskell Communities and Activities Report

http://tinyurl.com/haskcar

Thirtieth Edition — May 2016

Mihai Maruseac (ed.)

Chris Allen
Christopher Anand
Francesco Ariis
Heinrich Apfelmus
Emil Axelsson
Christiaan Baaij
Carl Baatz
Doug Beardsley
Ingo Blechschmidt
Emanuel Borsboom
Jeroen Bransen
Joachim Breitner
Björn Buckwalter
Erik de Castro Lopo
Manuel M. T. Chakravarty
Roman Cheplyaka
Olaf Chitil
Alberto Gómez Corona
Duncan Coutts
Tobias Dammers
Kei Davis
Atze Dijkstra
Corentin Dupont
Richard Eisenberg
Tom Ellis
Maarten Faddegon
Dennis Felsing
Julian Fleischer
Phil Freeman
Ben Gamari
PÁLI Gábor János
Michal J. Gajda
Andrew Gibiansky
Brett G. Giles
Andrew Gill
Mikhail Glushenkov
Mark Grebe
Daniel Gröber
Jurriaan Hage
Bastiaan Heeren
Joey Hess
Guillaume Hoffmann
Nicu Ionita
Bob Ippolito
Patrik Jansson
Robin KAY
Anton Kholomiov
Oleg Kiselyov
Edward Kmett
Balázs Kőműves
Eric Kow
Nickolay Kudasov
Rob Leslie
Ben Lippmeier
Andres Löh
Rita Loogen
Ian Lynagh
Douglas McClean
Gilberto Melfe
Simon Michael
Mantas Markevicius
Dino Morelli
Rishiyur Nikhil
Antonio Nikishaev
Ulf Norell
Ivan Perez
Jens Petersen
Simon Peyton Jones
Matthew Pickering
Gracjan Polak
Bryan Richter
Herbert Valerio Riedel
Jeffrey Rosenbluth
David Sabel
Martijn Schrage
Carter Tazio Schonwald
Tom Schrijvers
Michael Schröder
Austin Seipp
Jeremy Shaw
Christian Höner zu Siederdissen
Aditya Siram
Gideon Sireling
Michael Snoyman
Kyle Marek-Spartz
Lennart Spitzner
Henk-Jan van Tuyl
Bernhard Urban
Alessio Valentini
Daniel Wagner
Michael Walker
Greg Weber
Ingo Wechsung
Li-yao Xia
Kazu Yamamoto
Edward Z. Yang
Brent Yorgey
Alan Zimmerman
Marco Zocca

Preface

This is the 30th edition of the Haskell Communities and Activities Report.

Highlights of the edition include the announcement for the new Haskell' committee, plans for Haskell 2020 and GHC 8.0, and the first mention of the Stack tool.

Since the previous report, some new interesting entries were submitted but, sadly, we also had to remove old, stale items that were not updated in the last two years. Please send reports for them for the next edition to revive them. For this edition, we had around 10 entries which resurfaced and a large number received updates. We hope to see more entries revived and updated in the next edition.

As usual, fresh entries – either completely new or old entries which have been revived after a short temporarily disappearance – are formatted using a blue background, while updated entries have a header with a blue background.

A call for new HCAR entries and updates to existing ones will be issued on the Haskell mailing lists in late September/early October. As a simple statistic, for the last 7 editions, the average time between the call for updates for the HCAR and the day the new edition has been released was 42.2 days. That means that if the next call will arrive at start of October then the next edition of the report is due to appear mid November.

Now enjoy the current report and see what other Haskellers have been up to lately. Any feedback is very welcome, as always.

Mihai Maruseac, University of Massachusetts Boston, US
⟨hcar@haskell.org⟩

Contents

1 Community **6**
1.1 Haskell' — Haskell 2020 . . . . . **6**
1.2 Haskellers **6**

2 Books, Articles, Tutorials **7**
2.1 Oleg's Mini Tutorials and Assorted Small Projects **7**
2.2 School of Haskell **7**
2.3 Haskell Programming from first principles, a book forall **8**
2.4 Learning Haskell **8**

3 Implementations **9**
3.1 The Glasgow Haskell Compiler **9**
3.2 The Helium Compiler **12**
3.3 UHC, Utrecht Haskell Compiler **12**
3.4 Frege **13**
3.5 Specific Platforms **13**
3.5.1 Haskell on FreeBSD 13
3.5.2 Debian Haskell Group 13
3.5.3 Fedora Haskell SIG 14

4 Related Languages and Language Design **15**
4.1 Agda **15**
4.2 Disciple **15**

5 Haskell and . . . **16**
5.1 Haskell and Parallelism **16**
5.1.1 Eden 16
5.1.2 Auto-parallelizing Pure Functional Language System 17
5.2 Haskell and the Web **17**
5.2.1 WAI 17
5.2.2 Yesod 18
5.2.3 Warp 19
5.2.4 Mighttpd2 — Yet another Web Server 19
5.2.5 Happstack 19
5.2.6 Snap Framework 19
5.2.7 MFlow 19
5.2.8 JS Bridge 20
5.2.9 PureScript 21
5.3 Haskell and Compiler Writing **21**
5.3.1 MateVM 21
5.3.2 UUAG 21

6 Development Tools **23**
6.1 Environments **23**
6.1.1 ghc-mod — Happy Haskell Programming 23
6.1.2 haskell-ide-engine, a project for unifying IDE functionality 23
6.1.3 Haskell IDE From FP Complete 24
6.1.4 HaRe — The Haskell Refactorer 24
6.1.5 ghc-exactprint 24
6.1.6 Haskino 25
6.1.7 IHaskell: Haskell for Interactive Computing 26
6.1.8 Haskell for Mac 27

6.2 Code Management **27**
6.2.1 Darcs 27
6.2.2 `cab` — A Maintenance Command of Haskell Cabal Packages 28
6.3 Deployment **28**
6.3.1 Cabal 28
6.3.2 The Stack build tool 29
6.3.3 Stackage: the Library Dependency Solution 29
6.3.4 Haskell Cloud 30
6.4 Others **30**
6.4.1 ghc-heap-view 30
6.4.2 ghc-vis 30
6.4.3 Hat — the Haskell Tracer 31
6.4.4 Tasty 31
6.4.5 Generic random generators 32
6.4.6 Automatic type inference from JSON 32
6.4.7 Exference 32
6.4.8 Lentil 33
6.4.9 Hoed – The Lightweight Algorithmic Debugger for Haskell 33
6.4.10 Déjà Fu: Concurrency Testing 34
6.4.11 The Remote Monad Design Pattern 35

7 Libraries, Applications, Projects 36
7.1 Language Features **36**
7.1.1 Conduit 36
7.1.2 GHC type-checker plugin for kind Nat 36
7.1.3 Dependent Haskell 37
7.1.4 Yampa 37
7.2 Education **38**
7.2.1 Holmes, Plagiarism Detection for Haskell 38
7.2.2 Interactive Domain Reasoners 39
7.2.3 The Incredible Proof Machine 40
7.3 Mathematics, Numerical Packages and High Performance Computing **40**
7.3.1 hblas 40
7.3.2 Numerical 40
7.3.3 combinat 41
7.3.4 petsc-hs 41
7.4 Data Types and Data Structures **41**
7.4.1 Transactional Trie 41
7.4.2 fixplate 41
7.4.3 generics-sop 42
7.5 Databases and Related Tools **42**
7.5.1 Persistent 42
7.5.2 Riak bindings 42
7.5.3 Opaleye 43
7.5.4 HLINQ - LINQ for Haskell 43
7.5.5 YeshQL 43
7.6 User Interfaces **44**
7.6.1 HsQML 44
7.6.2 threepenny-gui 44
7.6.3 reactive-banana 45
7.6.4 fltkhs - GUI bindings to the FLTK library 45
7.6.5 wxHaskell 46
7.7 Graphics and Audio **46**
7.7.1 vect 46
7.7.2 diagrams 47
7.7.3 Chordify 48
7.7.4 csound-expression 48
7.7.5 hmidi 49

7.8 Text and Markup Languages . . . **49**
7.8.1 lhs2TEX . . . 49
7.8.2 pulp . . . 50
7.8.3 Unicode things . . . 50
7.8.4 Ginger . . . 50
7.9 Natural Language Processing . . . **51**
7.9.1 NLP . . . 51
7.9.2 GenI . . . 51
7.10 Embedding DSLs for Low-Level Processing . . . **52**
7.10.1 CλaSH . . . 52
7.10.2 Feldspar . . . 52
7.11 Games . . . **53**
7.11.1 EtaMOO . . . 53
7.11.2 scroll . . . 53
7.11.3 Nomyx . . . 54
7.11.4 Barbarossa . . . 54
7.12 Others . . . **55**
7.12.1 ADPfusion . . . 55
7.12.2 Generalized Algebraic Dynamic Programming . . . 55
7.12.3 leapseconds-announced . . . 57
7.12.4 hledger . . . 57
7.12.5 arbtt . . . 58
7.12.6 Transient . . . 58
7.12.7 tttool . . . 58
7.12.8 gipeda . . . 59
7.12.9 Octohat (Stack Builders) . . . 59
7.12.10 git-annex . . . 59
7.12.11 openssh-github-keys (Stack Builders) . . . 59
7.12.12 propellor . . . 60
7.12.13 dimensional: Statically Checked Physical Dimensions . . . 60
7.12.14 igrf: The International Geomagnetic Reference Field . . . 61
7.12.15 Haskell in Green Land . . . 61
7.12.16 Kitchen Snitch server . . . 62
7.12.17 DSLsofMath . . . 62

8 Commercial Users **63**
8.1 Well-Typed LLP . . . **63**
8.2 Bluespec Tools for Design of Complex Chips and Hardware Accelerators . . . **63**
8.3 Better . . . **64**
8.4 Keera Studios LTD . . . **64**
8.5 Stack Builders . . . **65**
8.6 Optimal Computational Algorithms, Inc. . . . **66**
8.7 Snowdrift.coop . . . **66**
8.8 McMaster Computing and Software Outreach . . . **67**

9 Research and User Groups **68**
9.1 Haskell at Eötvös Loránd University (ELTE), Budapest . . . **68**
9.2 Artificial Intelligence and Software Technology at Goethe-University Frankfurt . . . **68**
9.3 Functional Programming at the University of Kent . . . **69**
9.4 Haskell at KU Leuven, Belgium . . . **69**
9.5 HaskellMN . . . **70**
9.6 Functional Programming at KU . . . **70**
9.7 fp-syd: Functional Programming in Sydney, Australia . . . **70**
9.8 Regensburg Haskell Meetup . . . **71**
9.9 Curry Club Augsburg . . . **71**
9.10 Italian Haskell Group . . . **71**

1 Community

1.1 Haskell' — Haskell 2020

Report by:	Herbert Valerio Riedel
Participants:	Andres Löh, Antonio Nikishaev, Austin Seipp, Carlos Camarao de Figueiredo, Carter Schonwald, David Luposchainsky, Henk-Jan van Tuyl, Henrik Nilsson, Herbert Valerio Riedel, Iavor Diatchki, John Wiegley, José Manuel Calderón Trilla, Jurriaan Hage, Lennart Augustsson, M Farkas-Dyck, Mario Blažević, Nicolas Wu, Richard Eisenberg, Vitaly Bragilevsky, Wren Romano

Haskell' is an ongoing process to produce revisions to the Haskell standard, incorporating mature language extensions and well-understood modifications to the language. New revisions of the language are expected once per year.

The goal of the Haskell Language committee together with the Core Libraries Committee is to work towards a new Haskell 2020 Language Report. The Haskell Prime Process relies on *everyone* in the community to help by contributing proposals which the committee will then evaluate and, if suitable, help formalise for inclusion. Everyone interested in participating is also invited to join the haskell-prime mailing list.

Four years (or rather ~3.5 years) from now may seem like a long time. However, given the magnitude of the task at hand, to discuss, formalise, and implement proposed extensions (taking into account the recently enacted three-release-policy) to the Haskell Report, the process shouldn't be rushed. Consequently, this may even turn out to be a tight schedule after all. However, it's not excluded there may be an interim revision of the Haskell Report before 2020.

Based on this schedule, GHC 8.8 (likely to be released early 2020) would be the first GHC release to feature Haskell 2020 compliance. Prior GHC releases may be able to provide varying degree of conformance to drafts of the upcoming Haskell 2020 Report.

The Haskell Language 2020 committee starts out with 20 members which contribute a diversified skillset. These initial members also represent the Haskell community from the perspective of practitioners, implementers, educators, and researchers.

The Haskell 2020 committee is a language committee; it will focus its efforts on specifying the Haskell language itself. Responsibility for the libraries laid out in the Report is left to the Core Libraries Committee (CLC). Incidentally, the CLC still has an available seat; if you would like to contribute to the Haskell 2020 Core Libraries you are encouraged to apply for this opening.

1.2 Haskellers

Report by:	Michael Snoyman
Status:	experimental

Haskellers 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.

Haskellers is a web site in maintenance mode. No new features are being added, though the site remains active with many new accounts and job postings continuing. If you have specific feature requests, feel free to send them in (especially with pull requests!).

Haskellers remains a site intended for all members of the Haskell community, from professionals with 15 years experience to people just getting into the language.

Further reading

http://www.haskellers.com/

2 Books, Articles, Tutorials

2.1 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 three additions:

IO monad realized in 1965

The unabated debate about exactly how much category theory one needs to know to understand that strange beast of IO prompts a thought if monads, like related continuations,"are the things that are destined to be rediscovered time and time again," to borrow Reynolds phrase.

A 1994 paper on category theory monads and functional programming included an interesting historical side-note. It turns out that the essence of monads has been fully grasped back in 1965, by at least one person. That person has also discovered that imperative, control-flow-dependent computations can be embedded into calculus by turning control flow into data flow. That person is Peter Landin. His 1965 paper anticipated not only IO but also State and Writer monads, *call / cc*, streams and delayed evaluations, the relation of streams with co-routines, and even stream fusion. We revisit that paper.

Read the tutorial online.

Model checking of Functional Dependencies

Given multi-parameter type-class declarations with functional dependencies and a set of their instances, we explain how to check if the instances conform to a functional dependency. If the check fails we give a counter-example, which is more helpful than the compiler error messages. Our checker, which is a simple Prolog code, fills the real need nowadays: regrettably, GHC no longer does the functional dependency conformance check when the UndecidableInstances extension is on. The unconformant instances are admitted and cause problems, but at a later time and place and accompanied with even harder to understand error messages.

The method is model checking of the implication represented by the functional dependency. If the check fails, it produces a counter-example of a concrete set of instances that violate the dependency.

Read the tutorial online.

HSXML in tagless-final style

HSXML is a domain-specific language to write markup-style documents like XML and HTML, in the syntax closely resembling that of SXML:

```
(p "string" "string1" br "string3")
```

without unnecessary commas and other syntactic distractions. (It truly takes polyvariadic functions to the new level). Unlike SXML, HSXML is typed: it is statically ensured that a 'block-level element' like *p* cannot appear in the inline (i.e., character) content and that a character-content entity cannot appear in a pure element content. Context-polymorphism however lets us define, e.g., *title* to be either a block-level element or an attribute. The generated XML or HTML document is certainly well-formed; moreover, it will satisfy some validity constraints. HSXML is extensible: one can add more tags, more validity constraints, and more transformations. For example, the same *ChangeLog.hs* document may be rendered either as a web page or an *rss.xml* data feed.

The development of HSXML went through several iterations. The earlier, 2006 version, represented semi-structured data truly as a heterogeneous data structure: HList. Presently HSXML follows the 'final tagless' approach, and is reminiscent of the very first version, in which semi-structured data are represented as a monadic value polymorphic over the rendering monad. Instead of monad however, we now use monoid.

Read the tutorial online.

2.2 School of Haskell

Report by:	Michael Snoyman
Participants:	Edward Kmett, Simon Peyton Jones and others
Status:	active

The School of Haskell has been available since early 2013. It's main two functions are to be an education resource for anyone looking to learn Haskell and as a sharing resources for anyone who has built a valuable tutorial. The School of Haskell contains tutorials, courses, and articles created by both the Haskell community and the developers at FP Complete. Courses are available for all levels of developers.

Since the last HCAR, School of Haskell has been open sourced, and is available from its own domain name (schoolofhaskell.com). In addition, the underlying engine powering interactive code snippets, `ide-backend`, has also been released as open source.

Currently 3150 tutorials have been created and 441 have been officially published. Some of the most visited tutorials are *Text Manipulation*, *Attoparsec*, *Learning Haskell at the SOH*, *Introduction to Haskell - Haskell Basics*, and *A Little Lens Starter Tutorial*. Over the past year the School of Haskell has averaged about 16k visitors a month.

All Haskell programmers are encouraged to visit the School of Haskell and to contribute their ideas and projects. This is another opportunity to showcase the virtues of Haskell and the sophistication and high level thinking of the Haskell community.

Further reading

https://www.schoolofhaskell.com/

2.3 Haskell Programming from first principles, a book forall

Report by:	Chris Allen
Participants:	Julie Moronuki
Status:	In progress, content complete soon

Haskell Programming is a book that aims to get people from the barest basics to being well-grounded in enough intermediate Haskell concepts that they can self-learn what would be typically required to use Haskell in production or to begin investigating the theory and design of Haskell independently. We're writing this book because many have found learning Haskell to be difficult, but it doesn't have to be. What particularly contributes to the good results we've been getting has been an aggressive focus on effective pedagogy and extensive testing with reviewers as well as feedback from readers. My coauthor Julie Moronuki is a linguist who'd never programmed before learning Haskell and authoring the book with me.

Haskell Programming is currently content complete and is approximately 1,200 pages long in the v0.11.2 release. The book is available for sale during the early access, which includes the 1.0 release of the book in PDF. We're still editing and testing the material. We expect to release the book this summer.

Further reading

- http://haskellbook.com
- https://superginbaby.wordpress.com/2015/05/30/learning-haskell-the-hard-way/
- http://bitemyapp.com/posts/2015-08-23-why-we-dont-chuck-readers-into-web-apps.html

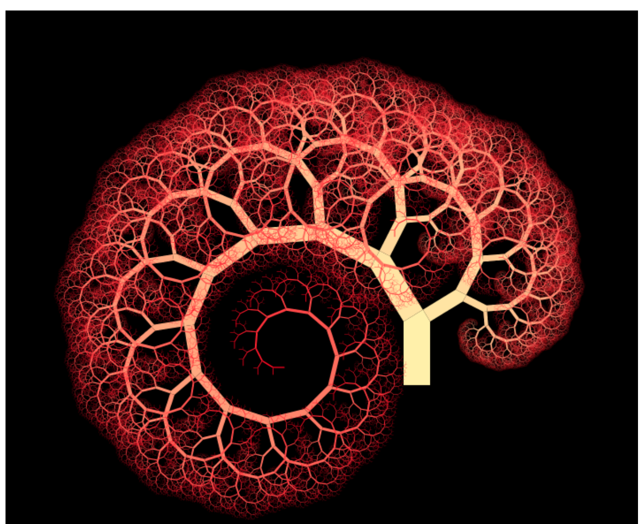
2.4 Learning Haskell

Report by:	Manuel M. T. Chakravarty
Participants:	Gabriele Keller
Status:	Work in progress with six published chapters

Learning Haskell is a new Haskell tutorial that integrates text and screencasts to combine in-depth explanations with the hands-on experience of live coding. It is aimed at people who are new to Haskell and functional programming. *Learning Haskell* does not assume previous programming expertise, but it is structured such that an experienced programmer who is new to functional programming will also find it engaging.

Learning Haskell combines perfectly with the Haskell for Mac programming environment, but it also includes instructions on working with a conventional command-line Haskell installation. It is a free resource that should benefit anyone who wants to learn Haskell.

Learning Haskell is still work in progress with six chapters already available. The current material covers all the basics including higher-order functions and the fundamentals of algebraic data types. *Learning Haskell* is approachable and fun - it includes topics such as illustrating various recursive structures using fractal graphics, such as this fractal tree.



Further chapters will be made available as we complete them.

Further reading

- *Learning Haskell* is free at http://learn.hfm.io
- Blog post with some background: http://blog.haskellformac.com/blog/learning-haskell

3 Implementations

3.1 The Glasgow Haskell Compiler

Report by: Ben Gamari
Participants: many others

GHC development churns onward — and **GHC 8.0 is right around the corner!** The final set of bugs are being fixed, and we hope to have a final release candidate, followed by the final release, in just a few weeks. More exciting developments await for 8.2 and beyond.

Major changes in GHC 8.0.1

GHC 8.0.1, the first release in the 8.0 series, will be released in May 2016. While this is significantly later than expected, we think that the features that this release brings should be well worth the wait. These include,

- **Lightweight, implicit call-stacks**[1]. Provides a `HasCallStack` constraint that can be added to any function to obtain a partial call-stack, with source locations, at that point in the program. `HasCallStack` and all related API functions are provided by the `GHC.Stack` module in `base`. The functions `error` and `undefined` now have a `HasCallStack` constraint and will print out the partial call-stack alongside the given error message.
- **Injective type families** (Wiki[2], paper[3]). Allows to annotate type families with injectivity information. Correctness of injectivity annotation is then verified by the compiler. Once compiler knows the annotation is correct it can us injectivity information during type checking.
- **Applicative do notation**. With the new `-XApplicativeDo` extension, GHC tries to desugar `do`-notation to `Applicative` where possible, giving a more convenient sugar for many common `Applicative` expressions. See the draft paper[4] and GHC Wiki[5] for details.
- A **beautiful new users guide**[6]. Now rewritten in reStructured Text, and with significantly improved output and documentation.
- **Visible type application**. This allows you to say, for example, `id @Bool` to specialize `id` to `Bool -> Bool`. With this feature, proxies are needed only in data constructors for pattern matching. Visible type patterns are due to be included sometime in the indeterminate future. See the Wiki[7] for details.
- **Kind Equalities**, which form the first step to building Dependent Haskell. This feature enables promotion of GADTs to kinds, kind families, heterogeneous equality (kind-indexed GADTs), and `* :: *`. See the Wiki[8] for more information.
- **Record system enhancements**. A new extension `DuplicateRecordFields` will be available in GHC 8.0, allowing multiple uses of the same field name with a very limited form of type-directed name resolution. Support for polymorphism over record fields is being worked on; another provisional new extension `OverloadedLabels` represents a first step in this process. See the GHC Wiki[9] for details.
- A huge improvement to **pattern match checking** (including much better coverage of GADTs), based on the work of Simon PJ and Georgios Karachalias. For more details, see their paper [10] with Tom Schrijvers and Dimitrios Vytiniotis. More information can be found in the GHC Wiki[11].
- **Custom type errors**, allowing library authors to offer more descriptive error messages than those offered by GHC. See the proposal[12] for more details.)
- **Improved generics representation** leveraging type-level literals. This makes `GHC.Generics` more expressive and uses new type system features to give more natural types to its representations.
- A new **DeriveLift language extension**, allowing the `Lift` type class from `Language.Haskell.TH.Syntax` to be derived automatically. This was introduced in the spirit of `DeriveDataTypeable` and `DeriveGeneric` to allow easier metaprogramming, and to allow users to easily define `Lift` instances without needing to depend on the existence of Template Haskell itself (see #1830).

[1] http://ghc.haskell.org/trac/ghc/wiki/ExplicitCallStack/ImplicitLocations
[2] http://ghc.haskell.org/trac/ghc/wiki/InjectiveTypeFamilies
[3] http://research.microsoft.com/en-us/um/people/simonpj/papers/ext-f/injective-type-families-acm.pdf
[4] http://research.microsoft.com/en-us/um/people/simonpj/papers/list-comp/applicativedo.pdf
[5] http://ghc.haskell.org/trac/ghc/wiki/ApplicativeDo
[6] http://downloads.haskell.org/~ghc/8.0.1-rc4/docs/html/users_guide/
[7] http://ghc.haskell.org/trac/ghc/wiki/ExplicitTypeApplication
[8] http://ghc.haskell.org/trac/ghc/wiki/DependentHaskell/Phase1
[9] http://ghc.haskell.org/trac/ghc/wiki/OverloadedRecordFields
[10] https://people.cs.kuleuven.be/~george.karachalias/papers/p424-karachalias.pdf
[11] http://ghc.haskell.org/trac/ghc/wiki/PatternMatchCheck
[12] http://ghc.haskell.org/trac/ghc/wiki/Proposal/CustomTypeErrors

- **Support for DWARF-based stacktraces**[13]. Haskell has at long last gained the ability to collect stack-traces of running programs. While still experimental, `base` now includes an interface which user code can use to request a representation of the current execution stack when running on a supported machine (currently Linux x86-64). Furthermore, the runtime system will now provide a backtrace of the currently running thread when thrown a `SIGUSR2` signal. Note that this functionality is highly experimental and there are some known issues which can potentially threaten the stability of the program.
- **Remote GHCi**[14]. The `-fexternal-interpreter` flag tells GHC to run interpreted code in a separate process. This provides various benefits, including allowing the interpreter to run profiled code (for example), thereby gaining access to stack traces[15] in GHCi.
- GHC now supports **environment files**. This is not any fundamental new capability but may prove to be a useful convenience. Build systems like Cabal call GHC with flags that define an (ephemeral) package environment, such as `-hide-all-packages -package-db=... -package this -package that`. An environment file lets the same information be stashed persistently in a file that GHC will pick up and use automatically. In principle this allows tools such as `Cabal` to generate an environment file and then you can use `ghc` or `ghci` directly and get the package environment of your project, rather than the default global environment. In addition to environments that live in a particular directory, it is possible to make a default global environment, or different global environments for different shell sessions. See the "Package environments" section of the GHC users manual[16] for more information.
- A new **Strict language extension**[17], allowing modules to be compiled such that local bindings are evaluated eagerly. Implemented by Adam Sandberg Eriksson based on an proposal by Johan Tibell.
- Significant improvements in cross-platform support, including a variety of fixes to **Windows linker support**, great improvements in **reliability on ARM** (GHC #11206[18] and others), revived **unregisterised m68k support**[19], and new support for **AIX targets** (Herbert) and Linux **PowerPC 64-bit big- and little-endian native code generation**.

[13]http://ghc.haskell.org/trac/ghc/wiki/DWARF
[14]http://ghc.haskell.org/trac/ghc/wiki/RemoteGHCi
[15]http://simonmar.github.io/posts/2016-02-12-Stack-traces-in-GHCi.html
[16]http://downloads.haskell.org/~ghc/8.0.1-rc4/docs/html/users_guide//packages.html#package-environments
[17]http://ghc.haskell.org/trac/ghc/wiki/StrictPragma
[18]https://ghc.haskell.org/trac/ghc/ticket/11206
[19]https://trofi.github.io/posts/191-ghc-on-m68k.html

- Improved support for **pattern synonyms**, including record syntax (GHC #8582[20]) and the ability to associate pattern synonyms with type constructors on export, implemented by Matthew Pickering. See Matthew's blog[21] for details.

Upcoming plans for GHC 8.2

With the super-major GHC 8.0 release out the door, plans have begun to form for the next major release, 8.2. Given that 8.0 saw a remarkable amount of churn, we hope to make the focus of 8.2 consolidation, stabilization, and optimization. For this reason, we hope you'll note there are relatively few "new" features in the lists below; instead we'd like to encourage contributors to polish, optimize, document, refactor, or finish the features we already have.

Of course, GHC only evolves because of its contributors. Please let us know if you have a pet project that you'd like to see merged!

Libraries, source language, type system

- **Indexed `Typeable` representations** (Ben Gamari, Simon Peyton Jones, et al). While GHC has long supported runtime type reflection through the `Typeable` typeclass, its current incarnation requires care to use, providing little in the way of type-safety. For this reason the implementation of types like `Data.Dynamic` must be implemented in terms of `unsafeCoerce` with no compiler verification.
  GHC 8.2 will address this by introducing indexed type representations, leveraging the type-checker to verify programs using type reflection. This allows facilities like `Data.Dynamic` to be implemented in a fully type-safe manner. See the paper [22] for an description of the proposal and the Wiki[23] for the current status of the implementation.
- Backpack[24] is targeting to be merged in GHC 8.2. More information to come. (Edward Z. Yang)
- Merge `Bifoldable` and `Bitraversable` into `base` (Edward Kmett, Ryan Scott)
- Generalize the `deriving` algorithms for `Eq`, `Functor`, etc. to be able to derive the data types in `Data.Functor.Classes` (`Eq1`, `Eq2`, etc.), `Bifunctor`, `Bifoldable`, and `Bitraversable` (Ryan Scott)
- Deriving strategies (Ryan Scott): grant users the ability to choose explicitly how a class should be `derived` (using a built-in algorithm, `GeneralizedNewtypeDeriving`, `DeriveAnyClass`, or otherwise), addressing #10598.

[20]https://ghc.haskell.org/trac/ghc/ticket/8582
[21]http://mpickering.github.io/posts/2015-12-12-pattern-synonyms-8.html
[22]http://research.microsoft.com/en-us/um/people/simonpj/papers/haskell-dynamic/
[23]https://ghc.haskell.org/trac/ghc/wiki/Typeable/BenGamari
[24]https://ghc.haskell.org/trac/ghc/wiki/Backpack

◦ Exhaustiveness checking for `EmptyCase`s, addressing #10746.

Back-end and runtime system

◦ Compact regions (Giovanni Campagna, Edward Yang, This runtime system feature allows a referentially "closed" set of heap objects to be collected into a "compact region", allowing cheaper garbage collection, heap-object sharing between processes, and the possibility of inexpensive serialization. See the patch[25] and paper[26] for more information.
◦ Refactoring and improvements to the cost-center profiler (Ben Gamari): Allow heap profiler samples to be directed to the GHC eventlog, allowing correlation with other program events, enabling easier analysis by tooling and eventual removal of the old, rather crufty `.hp` profile format.
◦ Further improvements to debugging information (Ben Gamari): There are still a number of outstanding issues with GHC's DWARF implementation, some of which even carry the potential to crash the runtime system during stacktrace collection. GHC 8.2 will hopefully have these issues resolved, allowing debugging information to be used by end-user code in production.
  With stable stack unwinding support comes a number of opportunities for new serial and parallel performance analysis tools (e.g. statistical profiling) and debugging. As GHC's debugging information improves, we expect to see tooling developed to support these applications. See the DWARF status page[27] for futher information.
◦ Support for NUMA systems (Simon Marlow, in progress[28]). The aim is to reduce the number of remote memory accesses for multi-socket systems that have a mixture of local and remote memory.
◦ Experimental changes to the scheduler (Simon Marlow, in progress[29]) that enable the number of threads used for GC to be lower than the `-N` setting.

Frontend, build system and miscellaneous changes

◦ New Shake-based build system, `hadrian`, will be merged. (Andrey Mokhov)
◦ The improved LLVM backend plan plan didn't make the cut for 8.0, but will for 8.2. See the GHC Wiki[30] for details. (Austin Seipp)
◦ Deterministic builds [31]. Given the same environment, file and flags produce ABI compatible binaries. (Bartosz Nitka, in-progress)

[25]https://phabricator.haskell.org/D1264
[26]http://ezyang.com/papers/ezyang15-cnf.pdf
[27]https://ghc.haskell.org/trac/ghc/wiki/DWARF/80Status
[28]https://github.com/simonmar/ghc/tree/numa
[29]https://github.com/simonmar/ghc/commit/7e05ec18b4eda8d97e37015d415e627353de6b50
[30]http://ghc.haskell.org/trac/ghc/wiki/ImprovedLLVMBackend
[31]http://ghc.haskell.org/trac/ghc/wiki/DeterministicBuilds

Development updates and acknowledgments

2015 has been a remarkable year for GHC development. Over the last twelve months the GHC repository gained nearly 2500 commits by over one hundred authors. Of these authors, nearly half are first-time contributors. At the time of writing alone alone there is over one dozen open and actively updated differentials on Phabricator. It seems fair to say that GHC's development community is stronger than ever.

We have been very lucky to have Thomas Miedema, who has devoted countless hours triaging bugs, cleaning up the build system, advising new contributors, and generally helping out in a multitude of ways which often go un-noticed. We all have benefited immensely from Thomas' hard work and generosity; thanks Thomas!

Another contributor deserving of recognition is Herbert Valerio Riedel, who meticulously handles many of the finer details of GHC development: over the past year Herbert has spent numerous weekends thinking through compatibility considerations for library and warning changes, managing GHC's interaction with its external core libraries, cleaning up GHC's build system, and maintaining his invaluable PPA repository for Ubuntu and Debian-based systems. Our releases wouldn't be nearly as smooth without Herbert's attention to detail.

On the Windows front, Tamar Christina has been doing amazing work cleaning up the many nooks that have gone unattended until now. His work on the runtime linker should mean that GHC 8.0 will be considerably more reliable when linking against many Windows libraries.

The past year has brought a number of new contributors: Ryan Scott and Michael Sloan have picked up various generics and Template Haskell projects, Andrew Farmer has contributed a number of fixes to the cost-centre profiler, and Bartosz Nitka has made numerous contributions improving compiler determinism. We also also saw the beginnings of some very interesting work from Ömer Sinan Ağacan, who is looking at teaching GHC to unpack sum types. Michael Walker, David Lupochainsky and Herbert Valerio Riedel have also started honing GHC's warnings system by both bringing consistency to the currently rather organic flags and making the messages themselves more informative. George Karachalias merged his full rewrite of the pattern match checker, which is now far more precise than GHC's previous implementation.

In recent years the growth of the Haskell community has required that we better develop our infrastructure for change management. This lead to the formation of the Core Libraries Committee, which is now in its third year. As such, we are now beginning to see some of the committee's efforts come to fruition. With GHC 8.0 progress was made on all three active proposals:

◦ `Semigroup-Monoid` proposal: the `Data.Semigroup` module is now available in `base` and there are now

opt-in warnings for missing `Semigroup` instances in preparation for the eventual addition of `Semigroup` as a superclass of `Monoid`

- `MonadFail` proposal: the `Control.Monad.Fail` module is available in `base` and a `-XMonadFailDesugaring` language extension has been introduced, allowing users to use the new class in `do` desugaring
- `ExpandFloating` proposal: `expm1`, `log1p`, `log1pexp`, `log1mexp` have been added to the `Floating` class with defaults

We are also excited to see the revitalization of the Haskell Prime process with the formation of the Haskell 2020 committee. This committee will attempt to formalize some of the better-understood GHC language extensions and introduce them into the language standard. We look forward to watching this process progress.

Finally, GHC HQ would like to thank Futureice GmbH for the donation of a Mac Mini to the project. This machine will be used for continuous integration and general testing on Mac OS X targets. Thanks Futureice!

Of course, GHC has also benefited from countless more contributors who we don't have room to acknowledge here. We'd like to thank everyone who has contributed patches, bug reports, code reviews, and discussion to the GHC community over the last year. GHC only improves through your efforts!

Further reading

- GHC website: http://haskell.org/ghc/
- GHC users guide: http://downloads.haskell.org/~ghc/master/users_guide/
- `ghc-devs` mailing list: https://mail.haskell.org/mailman/listinfo/ghc-devs

3.2 The Helium Compiler

Report by:	Jurriaan Hage
Participants:	Bastiaan Heeren

Helium is a compiler that supports a substantial subset of Haskell 98 (but, e.g., n+k patterns are missing). Type classes are restricted to a number of built-in type classes and all instances are derived. The advantage of Helium is that it generates novice friendly error feedback, including domain specific type error diagnosis by means of specialized type rules. Helium and its associated packages are available from Hackage. Install it by running `cabal install helium`. You should also `cabal install lvmrun` on which it dynamically depends for running the compiled code.

Currently Helium is at version 1.8.1. The major change with respect to 1.8 is that Helium is again well-integrated with the Hint programming environment that Arie Middelkoop wrote in Java. The jar-file for Hint can be found on the Helium website, which is located at http://www.cs.uu.nl/wiki/Helium. This website also explains in detail what Helium is about, what it offers, and what we plan to do in the near and far future.

A student has added parsing and static checking for type class and instance definitions to the language, but type inferencing and code generating still need to be added. Completing support for type classes is the second thing on our agenda, the first thing being making updates to the documentation of the workings of Helium on the website.

3.3 UHC, Utrecht Haskell Compiler

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

UHC is the Utrecht Haskell Compiler, supporting almost all Haskell98 features and most of Haskell2010, plus experimental extensions.

Status Current active development directly on UHC:

- Making intermediate Core language available as a compilable language on its own, used by an experimental Agda backend (Philipp Hausmann).
- The platform independent part of UHC has been made available via Hackage, as package "uhc-light" together with a small interpreter for Core files (Atze Dijkstra, interpreter still under development).
- Implementing static analyses (Tibor Bremer, Jurriaan Hage, in progress).
- Rework of the type system (Alejandro Serrano, Jurriaan Hage, just started).

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 (→ 5.3.2).

Further reading

- UHC Homepage: http://www.cs.uu.nl/wiki/UHC/WebHome
- UHC Github repository: https://github.com/UU-ComputerScience/uhc
- Attribute grammar system: http://www.cs.uu.nl/wiki/HUT/AttributeGrammarSystem

3.4 Frege

Report by:	Ingo Wechsung
Participants:	Dierk König, Mark Perry, Marimuthu Madasami, Sean Corfield, Volker Steiss and others
Status:	actively maintained

Frege is a Haskell dialect for the Java Virtual Machine (JVM). It covers essentially Haskell 2010, though there are some mostly insubstantial differences. Several GHC language extensions are supported, most prominently *higher rank types*.

As Frege wants to be a *practical* JVM language, interoperability with existing Java code is essential. To achieve this, it is not enough to have a foreign function interface as defined by Haskell 2010. We must also have the means to inform the compiler about existing data types (i.e. Java classes and interfaces). We have thus replaced the FFI by a so called *native interface* which is tailored for the purpose.

The compiler, standard library and associated tools like Eclipse IDE plugin, REPL (interpreter) and several build tools are in a usable state, and development is actively ongoing. The compiler is self hosting and has no dependencies except for the JDK.

In the growing, but still small community, a consensus developed last summer that existing differences to Haskell shall be eliminated. Ideally, Haskell source code could be ported by just compiling it with the Frege compiler. Thus, the ultimate goal is for Frege to become *the* Haskell implementation on the JVM.

Already, in the last months, some of the most offending differences have been removed: lambda syntax, instance/class context syntax, recognition of True and False as boolean literals, lexical syntax for variables and layout-mode issues. Frege now also supports code without module headers.

Frege is available under the BSD-3 license at the GitHub project page. A ready to run JAR file can be downloaded or retrieved through JVM-typical build tools like Maven, Gradle or Leiningen.

All new users and contributors are welcome!

Currently, we have a new version of code generation in alpha status. This will be the base for future interoperability with Java 8 and above.

In April, a community member submitted his masters thesis about implementation of a STM library for Frege.

Further reading

https://github.com/Frege/frege

3.5 Specific Platforms

3.5.1 Haskell on FreeBSD

Report by:	PÁLI Gábor János
Participants:	FreeBSD Haskell Team
Status:	ongoing

The FreeBSD Haskell Team is a small group of people who maintain Haskell software on all actively supported versions of FreeBSD. The primarily supported implementation is the Glasgow Haskell Compiler together with Haskell Cabal, although one may also find Hugs and NHC98 in the ports tree. FreeBSD is a Tier-1 platform for GHC (on both x86 and x86_64) starting from GHC 6.12.1, hence one can always download native vanilla binary distributions for each new release.

We have a developer (staging) repository for Haskell ports that currently features around 600 of many of the popular Cabal packages. Most of the updates committed to that repository are continuously integrated to the official ports tree on a regular basis. In result, the FreeBSD Ports Collection still offers many popular and important Haskell software: GHC 7.10.2, Gtk2Hs, wxHaskell, XMonad, Pandoc, Gitit, Yesod, Happstack, Snap, Agda (along with its standard library), git-annex, and so on – all of them are available on 9.3-RELEASE and 10.2-RELEASE. Note that we decided to abandon tracking Haskell Platform (although all its former components are still there as individual packages), instead we updated the packages to match their versions on Stackage (at end of August).

If you find yourself interested in helping us or simply want to use the latest versions of Haskell programs on FreeBSD, check out our development repository on GitHub (see below) where you can find the latest versions of the ports together with all the important pointers and information required for contacting or contributing.

Further reading

https://github.com/freebsd-haskell/ports

3.5.2 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. At the time of writing, we maintain 905 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 profiling enabled and the documentation packages register with the system-wide index.

The current stable Debian release ("jessie") provides the Haskell Platform 2013.2.0.0 and GHC 7.6.3, while in Debian unstable and testing we ship GHC 7.10.4. A GHC 8.0 prerelease is staged in the experimental distribution.

Debian users benefit from the Haskell ecosystem on 17 architecture/kernel combinations, including the non-Linux-ports KFreeBSD and Hurd.

Further reading

http://wiki.debian.org/Haskell

3.5.3 Fedora Haskell SIG

Report by:	Jens Petersen
Participants:	Ricky Elrod, Ben Boeckel, and others
Status:	active

The Fedora Haskell SIG works to provide good Haskell support in the Fedora Project Linux distribution.

The current stable Fedora 23 release still has GHC 7.8.4. The main change for Fedora 24 which should be released in June is Pandoc 1.16. For Fedora 25 we want to finally move to GHC 7.10.3. In the meantime there is a ghc-7.10.3 Fedora Copr repo available for Fedora and EPEL 7, and also one for GHC 8.0.1. There is also a Fedora Copr repo for stack.

At the time of writing we have 312 Haskell source packages in Fedora.

If you are interested in Fedora Haskell packaging, please join our mailing-list and the Freenode #fedora-haskell channel. You can also follow @fedorahaskell for occasional updates.

Further reading

- Homepage: http://fedoraproject.org/wiki/Haskell_SIG
- Mailing-list: https://lists.fedoraproject.org/archives/list/haskell-devel@lists.fedoraproject.org/
- Package list: https://admin.fedoraproject.org/pkgdb/packager/haskell-sig/
- Copr repos: https://copr.fedorainfracloud.org/coprs/petersen/

4 Related Languages and Language Design

4.1 Agda

Report by:	Ulf Norell
Participants:	Ulf Norell, Nils Anders Danielsson, Andreas Abel, Jesper Cockx, Makoto Takeyama, Stevan Andjelkovic, Jean-Philippe Bernardy, James Chapman, Dominique Devriese, Peter Divianszki, Fredrik Nordvall Forsberg, Olle Fredriksson, Daniel Gustafsson, Alan Jeffrey, Fredrik Lindblad, Guilhem Moulin, Nicolas Pouillard, Andrés Sicard-Ramírez 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.

Since the release of Agda 2.4.0 in June 2014 a lot has happened in the Agda project and community. For instance:

- There have been two Agda courses at the Oregon Programming Languages Summer School (OPLSS). In 2014 by Ulf Norell, and in 2015 by Peter Dybjer.
- Agda has moved to github: https://github.com/agda/agda.
- Agda 2.4.2 was released in September 2014, and the latest stable version is Agda 2.4.2.4, released in September 2015.
- The restriction of Agda to not use Streicher's Axiom K was proved correct by Jesper Cockx et al. in the ICFP 2014 paper *Pattern Matching without K*.
- Instance arguments are now powerful enough to emulate Haskell-style type classes.
- The reflection machinery has been extended, making it possible to define convenient reflection based tactics.
- Improved compiler performance, and a new backend targeting the Utrecht Haskell Compiler (UHC).

Release of Agda 2.4.4 is planned for early 2016.

Further reading

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

4.2 Disciple

Report by:	Ben Lippmeier
Participants:	Amos Robinson, Max Swadling, Kyle Van Berendonck, Jacob Stanley, Viktar Basharymau, Erik de Castro Lopo, Ben Lippmeier
Status:	experimental, active development

The Disciplined Disciple Compiler (DDC) is a research compiler used to investigate program transformation in the presence of computational effects. It compiles a family of strict functional core languages and supports region and effect typing. This extra information provides a handle on the operational behaviour of code that isn't available in other languages. Programs can be written in either a pure/functional or effectful/imperative style, and one of our goals is to provide both styles coherently in the same language.

What is new?

DDC is in an experimental, pre-alpha state, though parts of it do work. In March this year we released DDC 0.4.2, with the following new features:

- Added LLVM code generation for higher order functions.
- Added automatic insert of run and box casts.
- Added multi-module compilation.
- Added desugaring of guards.
- Added primitive for working with arrays of boxed values and vector of primitive unboxed values.
- Added first cut PHP code generator.
- Added case-of-known-constructor transform.
- Added clustering and rate inference for Core Flow language.
- Source programs now accept unicode lambdas and dumps of intermediate code use lambdas for both term and type binders.
- Removed deprecated Eval and Lite language fragments.

Further reading

http://disciple.ouroborus.net

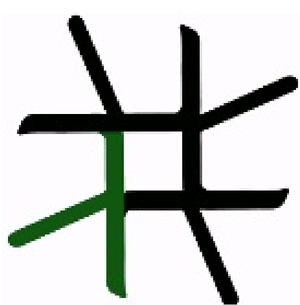
5 Haskell and ...

5.1 Haskell and Parallelism

5.1.1 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, Rita Loogen, Lukas Schiller, **in Sydney:** 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 primitive constructs are process abstractions and process instantiations. Higher-level coordination is achieved by defining *skeletons*, ranging from a simple parallel map to sophisticated master-worker schemes. They have been used to parallelize a set of non-trivial programs.

Eden's interface supports a simple definition of arbitrary communication topologies using *Remote Data*. The remote data concept can also be used to compose skeletons in an elegant and effective way, especially in distributed settings. A *PA*-monad enables the *eager* execution of user defined sequences of *Parallel Actions* in Eden.

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.

Tutorial: Rita Loogen: Eden - Parallel Functional Programming in Haskell, in: V. Zsók, Z. Horváth, and R. Plasmeijer (Eds.): CEFP 2011, Springer LNCS 7241, 2012, pp. 142-206. (see also: www.mathematik.uni-marburg.de/~eden/?content=cefp)

Implementation

Eden is implemented by modifications to the Glasgow-Haskell Compiler (extending its runtime system to use multiple communicating instances). Apart from MPI or PVM in cluster environments, Eden supports a shared memory mode on multicore platforms, which uses multiple independent heaps but does not depend on any middleware. Building on this runtime support, the Haskell package *edenmodules* defines the language, and *edenskels* provides a library of parallel skeletons.

A version based on GHC-7.8.2 (including binary packages and prepared source bundles) has been released in April 2014. This version fixed a number of issues related to error shut-down and recovery, and featured extended support for serialising Haskell data structures. The release of a version based on GHC-7.10 is in preparation. Previous stable releases with binary packages and bundles are still available on the Eden web pages.

The source code repository for Eden releases is james.mathematik.uni-marburg.de:8080/gitweb, the Eden libraries (Haskell-level) are also available via Hackage. Please contact us if you need any support.

Tools and libraries

The Eden trace viewer tool *EdenTV* provides a visualisation of Eden program runs on various levels. Activity profiles are produced for processing elements (machines), Eden processes and threads. In addition message transfer can be shown between processes and machines. EdenTV is written in Haskell and is freely available on the Eden web pages and on hackage. Eden's thread view can also be used to visualise ghc eventlogs. Recently, in the course of his Bachelor thesis, Bastian Reitemeier developed another trace viewer tool, *Eden-Tracelab*, which is capable of visualising large trace files, without being constrained by the available memory. Details can be found in his blogpost brtmr.de/2015/10/17/introducing-eden-tracelab.html.

The Eden skeleton library is under constant development. Currently it contains various skeletons for parallel maps, workpools, divide-and-conquer, topologies and many more. Take a look on the Eden pages.

Recent and Forthcoming Publications

- M. Dieterle: *Structured Parallelism by Composition - Design and implementation of a framework sup-*

porting skeleton compositionality, Doctoral Thesis, Philipps-Universität Marburg, February 2016, http://archiv.ub.uni-marburg.de/diss/z2016/0107/pdf/dmd.pdf.

- M. Dieterle, Th. Horstmeyer, R. Loogen, J. Berthold: *Skeleton Composition vs Stable Process Systems in Eden*, Journal of Functional Programming, to appear.
- J. Berthold, H.-W. Loidl, K. Hammond: *PAEAN: Portable Runtime Support for Physically-Shared-Nothing Architectures in Parallel Haskell Dialects*, Journal of Functional Programming, to appear.

Further reading

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

5.1.2 Auto-parallelizing Pure Functional Language System

Report by:	Kei Davis
Participants:	Dean Prichard, David Ringo, Loren Anderson, Jacob Marks
Status:	active

The main project goal is the demonstration of a light-weight, higher-order, polymorphic, pure functional language implementation in which we can experiment with automatic parallelization strategies, varying degrees of default function and constructor strictness. A secondary goal is to experiment with mechanisms for transparent fault tolerance.

We do not consider speculative or eager evaluation, or semantic strictness inferred by program analysis, so potential parallelism is dictated by the specified degree of default strictness and any strictness annotations.

Our approach is similar to that of the Intel Labs Haskell Research Compiler: to use GHC as a front-end to generate STG (or Core), then exit to our own back-end compiler. As in their case we do not attempt to use the GHC runtime. Our implementation is *light-weight* in that we are not attempting to support or recreate the vast functionality of GHC and its runtime. This approach is also similar to Don Stewart's except that we generate C instead of Java.

Current Status

Currently we have a fully functioning serial implementation and a primitive proof-of-design parallel implementation.

Immediate Plans

We are currently developing a more realistic parallel runtime. Bridging the gap between GHC STG (or Core) to our STG representation will be undertaken starting June 2016. An instrumentation framework will be developed in summer 2016.

Undergraduate/post-graduate Internships

If you are a United States citizen or permanent resident alien studying computer science or mathematics at the undergraduate level, or are a recent graduate, with strong interests in Haskell programming, compiler/runtime development, and pursuing a spring, fall, or summer internship at Los Alamos National Laboratory, this could be for you.

We don't expect applicants to necessarily already be highly accomplished Haskell programmers — such an internship is expected to be a combination of further developing your programming/Haskell skills and putting them to good use. If you're already a strong C hacker we could use that too.

The application process requires a bit of work so don't leave enquiries until the last day/month.

Term	Application Deadline
Summer 2016	Closed
Fall 2016	May 31, 2016
Spring 2017	Approx. July 2016
Summer 2017	Approx. January 2017
Fall 2017	Approx. May 2017

Email me at kei (at) lanl (dot) gov if interested in more information, and feel free to pass this along.

Further reading

A project web site is under construction.

5.2 Haskell and the Web

5.2.1 WAI

Report by:	Kazu Yamamoto
Participants:	Michael Snoyman, Greg Weber
Status:	stable

WAI (Web Application Interface) is an application interface between web applications and handlers in Haskell. The `Application` data type is defined as follows:

```
type Application
  =  Request
  -> (Response -> IO ResponseReceived)
  -> IO ResponseReceived
```

That is, a WAI application takes two arguments: a `Request` and a function to send a `Response`. So, the typical behavior of WAI application is processing a request, generating a response and passing the response to the function.

Historically speaking, this interface made possible to develop handlers other than HTTP. The WAI applications can run through FastCGI (wai-handler-fastcgi), run as stand-alone (wai-handler-webkit), etc. But the most popular handler is based on HTTP, of course. The major HTTP handler for WAI is Warp which now provides both HTTP/1.1 and HTTP/2. TLS (warp-tls) is also available. New transports such as WebSocket (wai-websocket) and Event Source (wai-extra) can be implemented, too.

It is possible to develop WAI applications directly. For instance, Hoogle and Mighttpd2 take this way. However, you may want to use web application frameworks such as Apiary, MFlow, rest, Servant, Scotty, Spock, Yesod, etc.

WAI also provides `Middleware`:

```
type Middleware = Application -> Application
```

WAI middleware can inspect and transform a request, for example by automatically gzipping a response or logging a request (wai-extra).

Since the last HCAR, WAI has successfully released version 3.2, which removes the experimental HTTP/2 module and some APIs. We bumped the version from 3.0 to 3.2 for consistency with Warp 3.2.

Further reading

- https://groups.google.com/d/forum/haskell-wai

5.2.2 Yesod

Report by:	Michael Snoyman
Participants:	Greg Weber, Luite Stegeman, Felipe Lessa
Status:	stable

Yesod is a traditional MVC RESTful framework. By applying Haskell's strengths to this paradigm, Yesod helps users create highly scalable web applications.

Performance scalablity comes from the amazing GHC compiler and runtime. GHC provides fast code and built-in evented asynchronous IO.

But Yesod is even more focused on scalable development. The key to achieving this is applying Haskell's type-safety to an otherwise traditional MVC REST web framework.

Of course type-safety guarantees against typos or the wrong type in a function. But Yesod cranks this up a notch to guarantee common web application errors won't occur.

- declarative routing with type-safe urls — say goodbye to broken links
- no XSS attacks — form submissions are automatically sanitized
- database safety through the Persistent library (→ 7.5.1) — no SQL injection and queries are always valid
- valid template variables with proper template insertion — variables are known at compile time and treated differently according to their type using the shakesperean templating system.

When type safety conflicts with programmer productivity, Yesod is not afraid to use Haskell's most advanced features of Template Haskell and quasi-quoting to provide easier development for its users. In particular, these are used for declarative routing, declarative schemas, and compile-time templates.

MVC stands for model-view-controller. The preferred library for models is Persistent (→ 7.5.1). Views can be handled by the Shakespeare family of compile-time template languages. This includes Hamlet, which takes the tedium out of HTML. Both of these libraries are optional, and you can use any Haskell alternative. Controllers are invoked through declarative routing and can return different representations of a resource (html, json, etc).

Yesod is broken up into many smaller projects and leverages Wai (→ 5.2.1) to communicate with the server. This means that many of the powerful features of Yesod can be used in different web development stacks that use WAI such as Scotty (→ 5.2.9) and Servant.

Yesod has been in API stability for some time. The 1.4 release was made in September of 2014, and we are still backwards-compatible to that. Even then, the 1.4 release was almost a completely backwards-compatible change. The version bump was mostly performed to break compatibility with older versions of dependencies, which allowed us to remove approximately 500 lines of conditionally compiled code. Notable changes in 1.4 include:

- New routing system with more overlap checking control.
- yesod-auth works with your database and your JSON.
- yesod-test sends HTTP/1.1 as the version.
- Type-based caching with keys.

The Yesod team is quite happy with the current level of stability in Yesod. Since the 1.0 release, Yesod has maintained a high level of API stability, and we intend to continue this tradition. Future directions for Yesod are now largely driven by community input and patches. We've been making progress on the goal of easier client-side interaction, and have high-level interaction with languages like Fay, TypeScript, and CoffeScript. GHCJS support is in the works.

The Yesod site (http://www.yesodweb.com/) is a great place for information. It has code examples, screencasts, the Yesod blog and — most importantly — a book on Yesod.

To see an example site with source code available, you can view Haskellers (→ 1.2) source code: (https://github.com/snoyberg/haskellers).

Further reading

http://www.yesodweb.com/

5.2.3 Warp

Report by:	Kazu Yamamoto
Participants:	Michael Snoyman
Status:	stable

Warp is a high performance, easy to deploy HTTP handler for WAI (→ 5.2.1). Its supports both HTTP/1.1 and HTTP/2.

Since the last HCAR, we have released Warp 3.2 which removes the experimental HTTP/2 APIs. The performance of HTTP/2 was drastically improved. The logic to handle static files such as `If-Modified-Since` was imported from Mighttpd2 (→ 5.2.4).

Further reading

- "Warp: A Haskell Web Server"
  - the May/June 2011 issue of IEEE Internet Computing
  - Issue page: http://www.computer.org/portal/web/csdl/abs/mags/ic/2011/03/mic201103toc.htm
  - PDF: http://steve.vinoski.net/pdf/IC-Warp_a_Haskell_Web_Server.pdf
- "Warp"
  - The Performance of Open Source Applications
  - HTML: http://www.aosabook.org/en/posa/warp.html

5.2.4 Mighttpd2 — Yet another Web Server

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

Mighttpd (called mighty) version 3 is a simple but practical Web server in Haskell. It provides features to handle static files, redirection, CGI, reverse proxy, reloading configuration files and graceful shutdown. Also TLS is supported.

The logic to handle static files has been transferred to Warp, an HTTP server library. So, Mighttpd became a simpler web application now.

You can install Mighttpd 3 (*mighttpd2*) from HackageDB. Note that the package name is *mighttpd2*, not *mighttpd3*, for historical reasons.

Further reading

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

5.2.5 Happstack

Report by:	Jeremy Shaw

Happstack is a diverse collection of libraries for creating web applications in Haskell. Libraries include support for type-safe routing, HTML templating, form validation, authentication and more.

In the last six months we have added two new experimental packages: `happstack-servant` and `happstack-websockets`. `happstack-servant` makes it easy to use Happstack with the new `servant` framework. `happstack-websockets` provides support for using `websockets`.

Further reading

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

5.2.6 Snap Framework

Report by:	Doug Beardsley
Participants:	Gregory Collins, Shu-yu Guo, James Sanders, Carl Howells, Shane O'Brien, Ozgun Ataman, Chris Smith, Jurrien Stutterheim, Gabriel Gonzalez, and others
Status:	active development

The Snap Framework is a web application framework built from the ground up for speed, reliability, stability, 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.

If you would like to contribute, get a question answered, or just keep up with the latest activity, stop by the `#snapframework` IRC channel on Freenode.

Further reading

- Snaplet Directory: http://snapframework.com/snaplets
- http://snapframework.com

5.2.7 MFlow

Report by:	Alberto Gómez Corona
Status:	active development

MFlow is a Web framework of the kind of other functional, stateful frameworks like WASH, Seaside, Ocsigen or Racket. MFlow does not use continuation passing properly, but a backtracking monad that permits the synchronization of browser and server and error tracing. This monad is on top of another "Workflow"

monad that adds effects for logging and recovery of process/session state. In addition, MFlow is RESTful. Any GET page in the flow can be pointed to with a REST URL.

The navigation as well as the page results are type safe. Internal links are safe and generate GET requests. POST request are generated when formlets with form fields are used and submitted. It also implements monadic formlets: They can modify themselves within a page. If JavaScript is enabled, the widget refreshes itself within the page. If not, the whole page is refreshed to reflect the change of the widget.

MFlow hides the heterogeneous elements of a web application and expose a clear, modular, type safe DSL of applicative and monadic combinators to create from multipage to single page applications. These combinators, called widgets or enhanced formlets, pack together javascript, HTML, CSS and the server code.

A paper describing the MFlow internals has been published in The Monad Reader issue 23.

Further reading

- MFlow as a DSL for web applications https://www.fpcomplete.com/school/to-infinity-and-beyond/older-but-still-interesting/MFlowDSL1
- MFlow, a continuation-based web framework without continuations http://themonadreader.wordpress.com/2014/04/23/issue-23
- How Haskell can solve the integration problem https://www.fpcomplete.com/school/to-infinity-and-beyond/pick-of-the-week/how-haskell-can-solve-the-integration-problem
- Towards a deeper integration: A Web language: http://haskell-web.blogspot.com.es/2014/04/towards-deeper-integration-web-language.html
- Perch https://github.com/agocorona/haste-perch
- hplayground demos http://tryplayg.herokuapp.com
- haste-perch-hplaygroun tutorial http://www.airpair.com/haskell/posts/haskell-tutorial-introduction-to-web-apps
- react.js a solution for a problem that Haskell can solve in better ways http://haskell-web.blogspot.com.es/2014/11/browser-programming-reactjs-as-solution.html
- MFlow demo site: http://mflowdemo.herokuapp.com

5.2.8 JS Bridge

Report by:	Tobias Dammers
Status:	Proprietary, with tentative plans for a free rewrite

For a recent project, we implemented a Haskell-JavaScript bridge that allows us to drive JavaScript functions running in a "real" server-side execution environment (node.js, phantomjs, or similar) while controlling the JavaScript code from within the Haskell host application using a monadic EDSL.

The use case for this was to simulate user interaction against arbitrary websites in the wild, in order to check these for compliance with various legal and other regulations. We did this by scripting a PhantomJS headless browser; the first version used JavaScript as the scripting language directly, but this soon proved to be cumbersome and brittle, so for the rewrite, we moved as much of the code as possible to Haskell.

The solution works as follows. First, the programmer defines a set of "calls", JavaScript functions to be called on the JavaScript side, by writing them in plain JavaScript, along with a pair of ‘Request‘ and ‘Response‘ types that represent the function's inputs and outputs. Then, a bit of boilerplate is added which generates a complete JavaScript script to run in the execution environment, such that it starts up an HTTP server that routes requests to the user-defined JavaScript functions. On the Haskell side, the execution environment is started in a subprocess, and function calls are delegated to HTTP requests. This has the added benefit of decoupling JavaScript asynchronous calls from Haskell parallelism, i.e., we can run things serially or in parallel on the Haskell side on a per-request basis without worrying about what is and is not asynchronous on the JavaScript side.

The final effect is that, once the wiring is in place, we can write code like the following:

```
withPhantom $ \p -> do
    openURL "http://www.google.com/" p
    waitForPageLoadComplete p
    injectClientSideScript clientScript p
    searchBox <- findSearchBox p
    setValue searchBox "cat pictures" p
    btn <- findSearchButton p
    clickOn btn p
    searchResults <- take 10 <$>
      getSearchResults p
    forM_ searchResults $ \result -> do
        liftIO $ print $
          searchResultTitle result
```

Needless to say, this is a lot nicer than the equivalent promises-ridden JavaScript code.

Since the library was written in an employment situation, and my employer has not agreed to releasing under a free software license, it is unfortunately not available to others; I do plan, however, to rewrite it from scratch in my own time, provided there is sufficient interest and/or a good use case on my side.

5.2.9 PureScript

Report by:	Phil Freeman
Status:	active, looking for contributors

PureScript is a small strongly typed programming language that compiles to efficient, readable JavaScript. The PureScript compiler is written in Haskell.

The PureScript language features Haskell-like syntax, type classes, rank-n types, extensible records and extensible effects.

PureScript features a comprehensive standard library, and a large number of other libraries and tools under development, covering data structures, algorithms, Javascript integration, web services, game development, testing, asynchronous programming, FRP, graphics, audio, UI implementation, and many other areas. It is easy to wrap existing Javascript functionality for use in PureScript, making PureScript a great way to get started with strongly-typed pure functional programming on the web. PureScript is currently used successfully in production in commercial code.

The PureScript compiler can be downloaded from purescript.org, or compiled from source from Hackage or Stackage.

Further reading

https://github.com/purescript/purescript/

5.3 Haskell and Compiler Writing

5.3.1 MateVM

Report by:	Bernhard Urban
Participants:	Harald Steinlechner
Status:	looking for new contributors

MateVM is a method-based Java Just-In-Time Compiler. That is, it compiles a method to native code on demand (i.e. on the first invocation of a method). We use existing libraries:

hs-java for processing Java Classfiles according to *The Java Virtual Machine Specification.*

harpy enables runtime code generation for `i686` machines in Haskell, in a domain specific language style.

We believe that Haskell is suitable to implement compiler technologies. However, we have to jump between "Haskell world" and "native code world", due to the low-level nature of Just-In-Time compiler in a virtual machine. This poses some challenges when it comes to signal handling and other interesting rather low level operations. Not immediately visible, the task turns out to be well suited for Haskell although we experienced some tensions with signal handling and GHCi. We are looking forward to sharing our experience on this.

In the current state we are able to execute simple Java programs. The compiler eliminates the JavaVM stack via abstract interpretation, does a liveness analysis, linear scan register allocation and finally machine code emission. The software architecture enables easy addition of further optimization passes based on an intermediate representation.

Future plans are, to add an interpreter to gather profile information for the compiler and also do more aggressive optimizations (e.g. method inlining or stack allocation). An interpreter can also be used to enable speculation during compilation and, if such a speculation fails, compiled code can *deoptimize* to the interpreter.

Apart from that, features are still missing to comply as a JavaVM, most noteable are proper support for classloaders, floating point operations or threads. We would like to see a real base library such as GNU Classpath or the JDK running with MateVM some day. Other hot topics are Hoopl and Garbage Collection.

We are looking for new contributors! If you are interested in this project, do not hesitate to join us on IRC (`#MateVM @ OFTC`) or contact us on Github.

Further reading

- https://github.com/MateVM
- http://docs.oracle.com/javase/specs/jvms/se7/html/
- http://hackage.haskell.org/package/hs-java
- http://hackage.haskell.org/package/harpy
- http://www.gnu.org/software/classpath/
- http://hackage.haskell.org/package/hoopl-3.8.7.4
- http://en.wikipedia.org/wiki/Club-Mate

5.3.2 UUAG

Report by:	Atze Dijkstra
Participants:	ST Group of Utrecht University
Status:	stable, maintained

UUAG is the *Utrecht University Attribute Grammar* system. It is a preprocessor for Haskell that makes it easy to write *catamorphisms*, i.e., functions that do to any data type what *foldr* does to lists. Tree walks are defined 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 auto-

matically. 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 (→ 3.3), the editor Proxima for structured documents (http://www.haskell.org/communities/05-2010/html/report.html#sect6.4.5), the Helium compiler (http://www.haskell.org/communities/05-2009/html/report.html#sect2.3), the Generic Haskell compiler, UUAG itself, and many master student projects. The current version is 0.9.52.1 (January 2015), is extensively tested, and is available on Hackage. There is also a Cabal plugin for easy use of AG files in Haskell projects.

We recently implemented the following enhancements:

Evaluation scheduling. We have done a project to improve the scheduling algorithms for AGs. The previously implemented algorithms for scheduling AG computations did not fully satisfy our needs; the code we write goes beyond the class of OAGs, but the algorithm by Kennedy and Warren (1976) results in an undesired increase of generated code due to non-linear evaluation orders. However, because we know that our code belongs to the class of linear orderable AGs, we wanted to find and algorithm that can find this linear order, and thus lies in between the two existing approaches. We have created a backtracking algorithm for this which is currently implemented in the UUAG (`-aoag` flag).

Another approach to this scheduling problem that we implemented is the use of SAT-solvers. The scheduling problem can be reduced to a SAT-formula and efficiently solved by existing solvers. The advantage is that this opens up possibilities for the user to influence the resulting schedule, for example by providing a cost-function that should be minimized. We have also implemented this approach in the UUAG which uses Minisat as external SAT-solver (`-loag` flag).

We have recently worked on the following enhancements:

Incremental evaluation. We have just finished a Ph.D. project that investigated incremental evaluation of AGs. The target of this work was to improve the UUAG compiler by adding support for incremental evaluation, for example by statically generating different evaluation orders based on changes in the input. The project has lead to several publications, but the result has not yet been implemented into the UUAG compiler.

Further reading

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

6 Development Tools

6.1 Environments

6.1.1 ghc-mod — Happy Haskell Programming

Report by:	Daniel Gröber
Status:	open source, actively developed

`ghc-mod` is both a backend program for enhancing editors and other kinds of development environments with support for Haskell, and an Emacs package providing the user facing functionality, internally called `ghc` for historical reasons. Other people have also developed numerous front ends for Vim and there also exist some for Atom and a few other proprietary editors.

This summer's two month `ghc-mod` hacking session was mostly spent (finally) getting a release supporting GHC 7.10 out the door as well as fixing bugs and adding full support for the *Stack* build tool.

Since the last report the *haskell-ide-engine* (→ 6.1.2) project has seen the light of day. There we are planning to adopt `ghc-mod` as a core component to use its environment abstraction.

The *haskell-ide-engine* project itself is aiming to be the central component of a unified Haskell Tooling landscape.

In the light of this `ghc-mod`'s mission statement remains the same but in the future it will be but one, important, component in a larger ecosystem of Haskell Tools.

We are looking forward to *haskell-ide-engine* making the Haskell Tooling landscape a lot less fragmented. However until this project produces meaningful results life goes on and `ghc-mod`'s ecosystem needs to be maintained.

Right now `ghc-mod` has only one core developer and a handful of occasional contributors. If *you* want to help make Haskell development even more fun come and join us!

Further reading

https://github.com/kazu-yamamoto/ghc-mod

6.1.2 haskell-ide-engine, a project for unifying IDE functionality

Report by:	Chris Allen
Participants:	Alan Zimmerman, Moritz Kiefer, Michael Sloan, Gracjan Polak, Daniel Gröber, others welcome
Status:	Open source, just beginning

haskell-ide-engine is a backend for driving the sort of features programmers expect out of IDE environments. *haskell-ide-engine* is a project to unify tooling efforts into something different text editors, and indeed IDEs as well, could use to avoid duplication of effort.

There is basic support for getting type information and refactoring, more features including type errors, linting and reformatting are planned. People who are familiar with a particular part of the chain can focus their efforts there, knowing that the other parts will be handled by other components of the backend. Integration for Emacs and Leksah is available and should support the current features of the backend. `haskell-ide-engine` also has a REST API with Swagger UI. Inspiration is being taken from the work the Idris community has done toward an interactive editing environment as well.

Help is very much needed and wanted so if this is a problem that interests you, please pitch in! This is not a project just for a small inner circle. Anyone who wants to will be added to the project on github, address your request to @alanz.

Further reading

- https://github.com/haskell/haskell-ide-engine
- https://mail.haskell.org/pipermail/haskell-cafe/2015-October/121875.html
- https://www.fpcomplete.com/blog/2015/10/new-haskell-ide-repo
- https://www.reddit.com/r/haskell/comments/3pt560/ann_haskellide_project/
- https://www.reddit.com/r/haskell/comments/3qbgmo/fp_complete_the_new_haskellide_repo/

6.1.3 Haskell IDE From FP Complete

Report by:	Michael Snoyman
Status:	available, stable

As of January of 2016, FP Complete™ has decomissioned FP Haskell Center (FPHC) in favor of other tooling. Specifically:

- School of Haskell (→ 2.2) continues to live on as an independent, open source project
- The underlying package set which was available in FP Haskell Center lives on via the Stackage project (→ 6.3.3) as LTS Haskell
- For reliable and user-friendly builds, we have shifted development focus towards the Stack build tool (→ 6.3.2)

Further reading

- https://www.fpcomplete.com
- https://www.schoolofhaskell.com
- http://haskellstack.com
- https://www.stackage.org
- https://github.com/fpco/ide-backend

6.1.4 HaRe — The Haskell Refactorer

Report by:	Alan Zimmerman
Participants:	Francisco Soares, Chris Brown, Stephen Adams, Huiqing Li, Matthew Pickering, Gracjan Polak

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 seventh major release. HaRe supports full Haskell 2010, and is integrated with (X)Emacs. 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.

Snapshots of *HaRe* are available from our GitHub repository (see below) and Hackage. There 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 on the University of Kent Refactoring Functional Programs page (see below).

There is also a Google+ community called HaRe, a Google Group called https://groups.google.com/forum/#!forum/hare and an IRC channel on freenode called *#haskell-refactorer*. IRC is the preferred contact method.

Current version of *HaRe* supports 7.10.2 and work is continuing to support GHC version 8.x forward. The new version makes use of *ghc-exactprint* (→ 6.1.5) library, which only has GHC support from GHC 7.10.2 onwards.

Development on the core *HaRe* is focusing is on making sure that deficiencies identified in the API Annotations in GHC used by *ghc-exactprint* are removed in time for GHC 8.0.1, so that the identity refactoring can cover more of the corner cases.

There is also a new *haskell-ide* project which will allow *HaRe* to operate as a plugin and will ease its integration into multiple IDEs.

Recent developments

- The current version is 8.2, which supports GHC 7.10.2 only, and was released in October 2015.
- Matthew Pickering has been deeply involved in the *ghc-exactprint* development, and successfully completed his Google Summer of Code project which involved bringing it up to standard, which has helped tremendously for *HaRe*.
- There is plenty to do, so anyone who has an interest is welcome to fork the repo and get stuck in.
- Stephen Adams is continuing his PhD at the University of Kent and will be working on data refactoring in Haskell.

Further reading

- http://www.cs.kent.ac.uk/projects/refactor-fp/
- https://github.com/RefactoringTools/HaRe
- https://github.com/alanz/ghc-exactprint
- http://mpickering.github.io/gsoc2015.html
- https://github.com/haskell/haskell-ide

6.1.5 ghc-exactprint

Report by:	Matthew Pickering
Participants:	Alan Zimmerman
Status:	Active, Experimental

`ghc-exactprint` aims to be a low-level foundation for refactoring tools. Unlike most refactoring tools, it works directly with the GHC API which means that it can understand any legal Haskell source file.

The program works in two phases. The first phase takes the output from the parser and converts all absolute source positions into relative source positions. This means that it is much easier to manipulate the AST as you do not have to worry about updating irrelevant parts of your program. The second phase performs the reverse process, it converts relative source

positions back into absolute positions before printing the source file. The entire library is based around a free monad which keeps track of which annotations should be where. Each process is then a different interpretation of this structure.

In theory these two processes should be entirely separate but at the moment they are not entirely decoupled due to shortcomings we are addressing in GHC 8.0.

In order to verify our foundations, the program has been run on every source file on Hackage. This testing highlighted a number of bugs which have been fixed for GHC 7.10.2. Apart from a few outstanding issues with very rare cases, we can now confidently say that `ghc-exactprint` is capable of processing *any* Haskell source file.

Over the last few months Alan Zimmerman has integrated `ghc-exactprint` into HaRe(→ 6.1.4) whilst Matthew Pickering participated in Google Summer of Code to provide integration with `HLint`.

Both of these proceeded smoothly, and are now working.

`ghc-exactprint` has also been used for a proof of concept tool to migrate code forward for AMP and MRP, see link below.

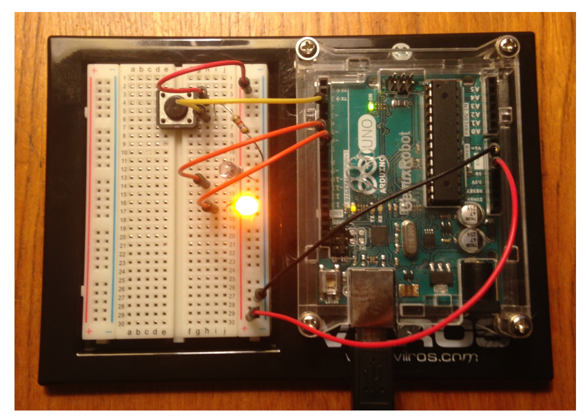
Alan Zimmerman also presented `ghc-exactprint` at HIW2015, and Matthew Pickering at SkillsMatter in October. Links to the respective videos are provided below.

Further reading

- https://github.com/alanz/ghc-exactprint
- https://github.com/hvr/Hs2010To201x
- https://www.youtube.com/watch?v=U5_9mfQAUBo - HIW2015
- https://skillsmatter.com/skillscasts/6539-a-new-foundation-for-refactoring-ghc-exactprint - Skills Matter, (free) registration required

6.1.6 Haskino

Report by:	Andrew Gill
Participants:	Mark Grebe
Status:	active



Haskino is a Haskell development environment for programming the Arduino microcontroller boards in a high level functional language instead of the low level C language normally used.

This work started with Levent Erkök's hArduino package. The original version of Haskino, extended hArduino by applying the concepts of the strong remote monad design pattern to provide a more efficient way of communicating, and generalizing the controls over the remote execution. In addition, it added a deep embedding, control structures, an expression language, and a redesigned firmware interpreter to enable standalone software for the Arduino to be developed using the full power of Haskell.

The current version of Haskino continues to build on this work. Haskino is now able to directly generate C programs from our Arduino Monad. This allows the same monadic program to be quickly developed and prototyped with the interpreter, then compiled to C for more efficient operation. In addition, we have added scheduling capability with lightweight threads and semaphores for inter-thread synchronization.

The development has been active over the past year. A paper was published at PADL 2016 for original version, and there is a paper accepted for presentation at TFP 2016 for the new scheduled and compiled version.

Further reading

- https://github.com/ku-fpg/haskino
- https://github.com/ku-fpg/wiki

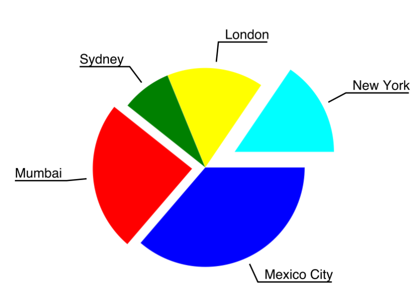
6.1.7 IHaskell: Haskell for Interactive Computing

Report by:	Andrew Gibiansky
Status:	stable

IHaskell is an interactive interface for Haskell development. It provides a *notebook* interface (in the style of Mathematica or Maple). The notebook interface runs in a browser and provides the user with editable cells in which they can create and execute code. The output of this code is displayed in a rich format right below, and if it's not quite right, the user can go back, edit the cell, and re-execute. This rich format defaults to the same boring plain-text output as GHCi would give you; however, library authors will be able to define their own formats for displaying their data structures in a useful way, with the only limit being that the display output must be viewable in a browser (images, HTML, CSS, Javascript). For instance, integration with graphing libraries yields in-browser data visualizations, while integration with Aeson's JSON yields a syntax-highlighted JSON output for complex data structures.

```
toRenderable
  $ pie_title .~ "Relative Population"
  $ pie_plot . pie_data .~ map pitem values
  $ def
```

Relative Population



Implementation-wise, IHaskell is a language kernel backend for the Jupyter project, a *language-agnostic* protocol and set of frontends by which interactive code environments such as REPLs and notebooks can communicate with a language evaluator backend. IHaskell also provides a generic library for writing Jupyter kernels, which has been used successfully in the ICryptol project.

```
-- We can draw diagrams, right in the notebook.
:extension NoMonomorphismRestriction
import Diagrams.Prelude

-- By Brent Yorgey
-- Draw a Sierpinski triangle!
sierpinski 1 = eqTriangle 1
sierpinski n =     s
                  ===
               (s ||| s) # centerX
  where s = sierpinski (n-1)

-- The `diagram` function is used to display them
diagram $ sierpinski 4
            # centerXY
            # fc black
          `atop` square 10
                   # fc white
```

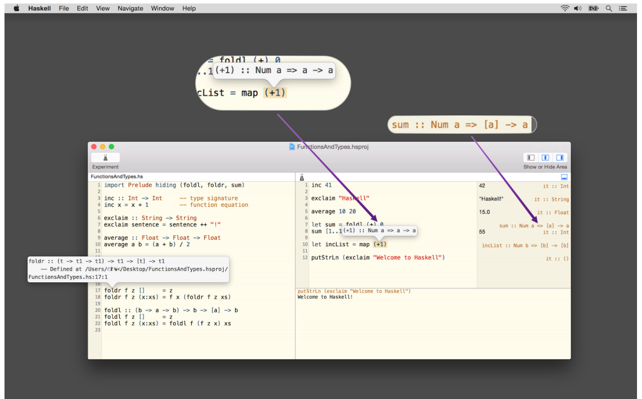
Integration with popular Haskell libraries can give us beautiful and potentially interactive visualizations of Haskell data structures. On one hand, this could range from simple things such as foldable record structures — imagine being able to explore complex nested records by folding and unfolding bits and pieces at a time, instead of trying to mentally parse them from the GHCi output. On the other end, we have interactive outputs, such as Parsec parsers which generate small input boxes that run the parser on any input they're given. And these things are just the beginning — tight integration with IPython may eventually be able to provide things such as code-folding in your REPL or an integrated debugger interface.

Further reading

https://github.com/gibiansky/IHaskell

6.1.8 Haskell for Mac

Report by:	Manuel M. T. Chakravarty
Status:	Available & actively developed



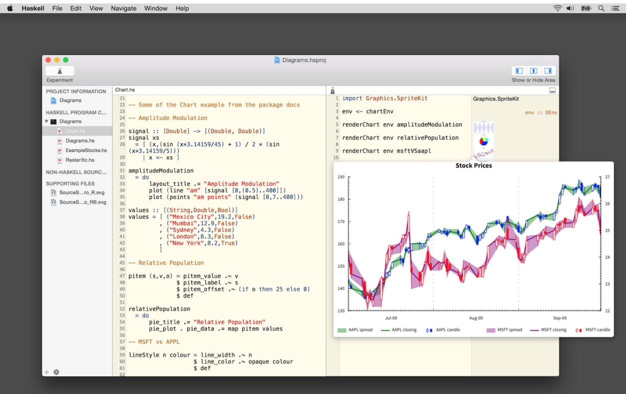
Haskell for Mac is an easy-to-use integrated programming environment for Haskell on OS X. It includes its own Haskell distribution and requires no further set up. It features interactive Haskell playgrounds to explore and experiment with code. Playground code is not only type-checked, but also executed while you type, which leads to a fast turn around during debugging or experimenting with new code.

Integrated environment. Haskell for Mac integrates everything needed to start writing Haskell code, including an editor with syntax highlighting and smart identifier completion. Haskell for Mac creates Haskell projects based on standard Cabal specifications for compatibility with the rest of the Haskell ecosystem. It includes the Glasgow Haskell Compiler (GHC) and over 200 of the most popular packages of LTS Haskell package sets. Matching command line tools and extra packages can be installed, too.

Type directed development. Haskell for Mac uses GHC's support for deferred type errors so that you can still execute playground code in the face of type errors. This is convenient during refactoring to test changes, while some code still hasn't been adapted to new signatures. Moreover, you can use type holes to stub out missing pieces of code, while still being able to run code. The system will also report the types expected for holes and the types of the available bindings.

Interactive HTML, graphics & animation. Haskell for Mac comes with support for web programming, network programming, graphics programming, animations, and much more. Interactively generate web pages, charts, animations, or even games (with the OS X SpriteKit support). Graphics are also live and change as you modify the program code.

Haskell for Mac is made for beginners and experts alike. The continuous feedback of interactive Haskell playgrounds is ideal for learning functional programming. At the same time, Haskell playgrounds provide the ideal environment for experts to quickly experiment with new ideas and to iterate on prototype code.



Haskell for Mac is available for purchase from the Mac App Store. Just search for "Haskell", or visit our website for a direct link. We are always available for questions or feedback at support@haskellformac.com.

The current version of Haskell for Mac is based on GHC 7.10.3 and LTS Haskell 5.9. Haskell for Mac tracks new GHC and LTS Haskell releases with a bias towards stability and ease of use.

Further reading

The Haskell for Mac website: http://haskellformac.com

6.2 Code Management

6.2.1 Darcs

Report by:	Guillaume Hoffmann
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.

One year, two sprints and 440 patches after the 2.10 release, we have released Darcs 2.12 (April 2016). This new major release includes the new `darcs show dependencies` command (for exporting the patch dependencies graph of a repository to the Graphviz format), improvements for Git import, and improvements to `darcs whatsnew` to facilitate support of Darcs by third-party version control front ends.

SFC and donations Darcs is free software licensed under the GNU GPL (version 2 or greater). 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
- http://darcs.net/Releases/2.12

6.2.2 cab — A Maintenance Command of Haskell Cabal Packages

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

`cab` is a MacPorts-like maintenance command of Haskell cabal packages. Some parts of this program are a wrapper to `ghc-pkg` and `cabal`.

If you are always confused due to inconsistency of `ghc-pkg` and `cabal`, or if you want a way to check all outdated packages, or if you want a way to remove outdated packages recursively, this command helps you.

`cab` now supports Cabal version 23 and `cab delete` got safer on Windows.

Further reading

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

6.3 Deployment

6.3.1 Cabal

Report by:	Mikhail Glushenkov
Status:	Stable, actively developed

Background

Cabal is the standard packaging system for Haskell software. It specifies a standard way in which Haskell libraries and applications can be packaged so that it is easy for consumers to use them, or re-package them, regardless of the Haskell implementation or installation platform.

`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

We've just released versions 1.24 of Cabal and `cabal-install`. 1.24 incorporates more than a thousand commits by 89 different contributors. Main user-visible changes in this release are:

- Nix-style local builds in `cabal-install` (so far only a technical preview). See this post by Edward Z. Yang for more details.

- Integration of a new security scheme for Hackage based on The Update Framework. So far this is not enabled by default, pending some changes on the Hackage side. See these three posts by Edsko de Vries and Duncan Coutts for more information.

- Support for specifying setup script dependencies in `.cabal` files. See this post by Duncan Coutts for more information.

- Support for HTTPS downloads in `cabal-install`. HTTPS is now used by default for downloads from Hackage.

- `cabal upload` learned how to upload documentation to Hackage (`cabal upload --doc`).

- In related news, `cabal haddock` now can generate documentation intended for uploading to Hackage (`cabal haddock --for-hackage`). `cabal upload --doc` runs this command automatically if the documentation for current package wasn't generated yet.

- New `cabal-install` command: `gen-bounds`. See here for more information.

- It's now possible to limit the scope of `--allow-newer` to single packages in the install plan, both on the command line and in the config file. See here for an example.

- New `cabal user-config` subcommand: `init`, which creates a default `~/.cabal/config` file.

- New config file field: `extra-framework-dirs` (extra locations to find OS X frameworks in).

- `cabal-install` solver now takes information about extensions and language flavours into account.

- New `cabal-install` option: `--offline`, which prevents `cabal-install` from downloading anything from the Internet.

- New `cabal upload` option `-P/--password-command` for reading Hackage password from arbitrary program output.

- Support for GHC 8 (NB: old versions of Cabal won't work with this version of GHC).

Full list of changes in Cabal 1.24 is available here; full list of changes in `cabal-install` 1.24 is available here.

Looking Forward

We plan to make a new release of Cabal/cabal-install approximately 6 months after 1.24 – that is, in late October or early November 2016. Main features that are currently targeted at 1.26 are:

- Further work on nix-style local builds, perhaps making that code path the default.
- Enabling Hackage Security by default.
- Native suport for "foreign libraries": Haskell libraries that are intended to be used by non-Haskell code.
- New Parsec-based parser for .cabal files.

We would like to encourage people considering contributing to take a look at the bug tracker on GitHub, take part in discussions on tickets and pull requests, or submit their own. The bug tracker is reasonably well maintained and it should be relatively clear to new contributors what is in need of attention and which tasks are considered relatively easy. For more in-depth discussion there is also the cabal-devel mailing list.

Further reading

- Cabal homepage: https://www.haskell.org/cabal/
- Cabal on GitHub: https://github.com/haskell/cabal

6.3.2 The Stack build tool

Report by:	Emanuel Borsboom
Status:	stable

Stack is a modern, cross-platform build tool for Haskell code. It is intended for Haskellers both new and experienced.

Stack handles the management of your toolchain (including GHC - the Glasgow Haskell Compiler - and, for Windows users, MSYS), building and registering libraries, building build tool dependencies, and more. While it can use existing tools on your system, Stack has the capacity to be your one-stop shop for all Haskell tooling you need.

The primary design point is reproducible builds. If you run `stack build` today, you should get the same result running `stack build` tomorrow. There are some cases that can break that rule (changes in your operating system configuration, for example), but, overall, Stack follows this design philosophy closely. To make this a simple process, Stack uses curated package sets called snapshots.

Stack has also been designed from the ground up to be user friendly, with an intuitive, discoverable command line interface.

Since its first release in June 2015, many people are using it as their primary Haskell build tool, both commercially and as hobbyists. New features and refinements are continually being added, with regular new releases.

Binaries and installers/packages are available for common operating systems to make it easy to get started. Download it at http://haskellstack.org/.

Further reading

http://haskellstack.org/

6.3.3 Stackage: the Library Dependency Solution

Report by:	Michael Snoyman
Status:	new

Stackage began in November 2012 with the mission of making it possible to build stable, vetted sets of packages. The overall goal was to make the Cabal experience better. Two years into the project, a lot of progress has been made and now it includes both Stackage and the Stackage Server. To date, there are over 1900 packages available in Stackage. The official site is https://www.stackage.org.

The Stackage project consists of many different components, linked to from the Stackage Github repository https://github.com/fpco/stackage#readme. These include:

- Stackage Nightly, a daily build of the Stackage package set
- LTS Haskell, which provides major-version compatibility for a package set over a longer period of time
- Stackage Server, which runs on stackage.org and provides browsable docs, reverse dependencies, and other metadata on packages
- Stackage Curator, a tool for running the various builds

The Stackage package set has first-class support in the Stack build tool (→ 6.3.2). There is also support for cabal-install via cabal.config files, e.g. https://www.stackage.org/lts/cabal.config.

There are dozens of individual maintainers for packages in Stackage. Overall Stackage curation is handled by the "Stackage curator" team, which consists of Michael Snoyman, Adam Bergmark, Dan Burton, and Jens Petersen.

Stackage provides a well-tested set of packages for end users to develop on, a rigorous continuous-integration system for the package ecosystem, some basic guidelines to package authors on minimal package compatibility, and even a testing ground for new versions of GHC. Stackage has helped encourage package

authors to keep compatibility with a wider range of dependencies as well, benefiting not just Stackage users, but Haskell developers in general.

If you've written some code that you're actively maintaining, don't hesitate to get it in Stackage. You'll be widening the potential audience of users for your code by getting your package into Stackage, and you'll get some helpful feedback from the automated builds so that users can more reliably build your code.

6.3.4 Haskell Cloud

Report by:	Gideon Sireling

Haskell Cloud is an OpenShift cartridge for deploying Haskell on Red Hat's open source PaaS cloud. It includes GHC 7.10, cabal-install, Gold linker, and a choice of pre-installed frameworks - a full list can be viewed on the Wiki.

Using a Haskell Cloud cartridge, existing Haskell projects can be uploaded, build, and run from the cloud with minimal changes. Ongoing development is focused on OpenShift v3 and GCH 8.

Further reading

- https://bitbucket.org/accursoft/haskell-cloud
- http://www.haskell.org/haskellwiki/Web/Cloud#OpenShift
- https://blog.openshift.com/functional-programming-in-the-cloud-how-to-run-haskell-on-openshift/

6.4 Others

6.4.1 ghc-heap-view

Report by:	Joachim Breitner
Participants:	Dennis Felsing
Status:	active development

The library ghc-heap-view provides means to inspect the GHC's heap and analyze the actual layout of Haskell objects in memory. This allows you to investigate memory consumption, sharing and lazy evaluation.

This means that the actual layout of Haskell objects in memory can be analyzed. You can investigate sharing as well as lazy evaluation using ghc-heap-view.

The package also provides the GHCi command `:printHeap`, which is similar to the debuggers' `:print` command but is able to show more closures and their sharing behaviour:

```
> let x = cycle [True, False]
> :printHeap x
_bco
> head x
True
> :printHeap x
let x1 = True : _thunk x1 [False]
in x1
> take 3 x
[True,False,True]
> :printHeap x
let x1 = True : False : x1
in x1
```

The graphical tool ghc-vis (→ 6.4.2) builds on ghc-heap-view.

Since version 0.5.3, ghc-heap-view also supports GHC 7.8.

Further reading

- http://www.joachim-breitner.de/blog/archives/548-ghc-heap-view-Complete-referential-opacity.html
- http://www.joachim-breitner.de/blog/archives/580-GHCi-integration-for-GHC.HeapView.html
- http://www.joachim-breitner.de/blog/archives/590-Evaluation-State-Assertions-in-Haskell.html

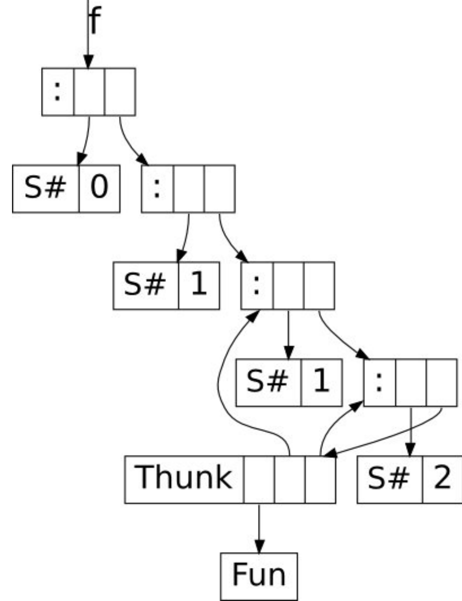
6.4.2 ghc-vis

Report by:	Joachim Breitner
Status:	active development

The tool ghc-vis visualizes live Haskell data structures in GHCi. Since it does not force the evaluation of the values under inspection it is possible to see Haskell's lazy evaluation and sharing in action while you interact with the data.

Ghc-vis supports two styles: A linear rendering similar to GHCi's `:print`, and a graph-based view where closures in memory are nodes and pointers between them are edges. In the following GHCi session a partially evaluated list of fibonacci numbers is visualized:

```
> let f = 0 : 1 : zipWith (+) f (tail f)
> f !! 2
> :view f
```



At this point the visualization can be used interactively: To evaluate a thunk, simply click on it and immediately see the effects. You can even evaluate thunks which are normally not reachable by regular Haskell code.

Ghc-vis can also be used as a library and in combination with GHCi's debugger.

Further reading

http://felsin9.de/nnis/ghc-vis

6.4.3 Hat — the Haskell Tracer

Report by:	Olaf Chitil

Hat is a source-level tracer for Haskell. Hat gives access to detailed, otherwise invisible information about a computation.

Hat helps locating errors in programs. Furthermore, it is useful for understanding how a (correct) program works, especially for teaching and program maintenance. Hat is not a time or space profiler. Hat can be used for programs that terminate normally, that terminate with an error message or that terminate when interrupted by the programmer.

You trace a program with Hat by following these steps:

1. With *hat-trans* translate all the source modules of your Haskell program into tracing versions. Compile and link (including the Hat library) these tracing versions with ghc as normal.

2. Run the program. It does exactly the same as the original program except for additionally writing a trace to file.

3. After the program has terminated, view the trace with a tool. Hat comes with several tools for selectively viewing fragments of the trace in different ways: *hat-observe* for Hood-like observations, *hat-trail* for exploring a computation backwards, *hat-explore* for freely stepping through a computation, *hat-detect* for algorithmic debugging, . . .

Hat is distributed as a package on Hackage that contains all Hat tools and tracing versions of standard libraries. Hat works with the Glasgow Haskell compiler for Haskell programs that are written in Haskell 98 plus a few language extensions such as multi-parameter type classes and functional dependencies. Note that all modules of a traced program have to be transformed, including trusted libraries (transformed in trusted mode). For portability all viewing tools have a textual interface; however, many tools require an ANSI terminal and thus run on Unix / Linux / OS X, but not on Windows.

In the longer term we intend to transfer the lightweight tracing technology that we use in Hoed also to Hat.

Further reading

- Initial website: http://projects.haskell.org/hat
- Hackage package:
  http://hackage.haskell.org/package/hat

6.4.4 Tasty

Report by:	Roman Cheplyaka
Participants:	Michael LaCorte, Sergey Vinokurov, and many others
Status:	actively maintained

Tasty is a modern testing framework for Haskell. As of May 2015, 230 hackage packages use Tasty for their tests. We've heard from several companies that use Tasty to test their Haskell software.

What's new since the last HCAR?

- Tasty now sets the number of parallel running tests equal to the number of available capabilities (i.e. the number set by `-N`) by default. As always, that can be changed with `-j`.
- Printing test results on Windows used to be slow, but now it's fast!
- Tasty-HUnit now has a new function, `testCaseSteps`, which lets you annotate a multi-step unit test. Here's an example:
  ```
  main =
    defaultMain $
    testCaseSteps "Multi-step test" $
    \step -> do

    step "Step 1"
    -- do something

    step "Step 2"
    -- do something else
  ```
  As a reminder from the last HCAR, Tasty-HUnit no longer uses the original HUnit package; instead it reimplements the relelvant subset of its API.
- The way Tasty-Golden works internally has changed. There are a few consequences (see the CHANGELOG for details); an interesting one is that you can now update golden files in parallel.
  Also, if a golden file doesn't exist, it will be created automatically. You'll see a message like
  ```
  UnboxedTuples:                     OK (0.04s)
    Golden file did not exist; created
  ```
  This is convenient when adding new tests.

Further reading

- For more information about Tasty and how to use it, please consult the README at

http://bit.ly/tasty-home

- Tasty has a mailing list http://bit.ly/tasty-ml and an IRC channel (`#tasty` on FreeNode), where you can get help with Tasty.

6.4.5 Generic random generators

Report by:	Li-yao Xia
Status:	Experimental, active development

The generic-random library automatically derives random generators for most datatypes. It can be used in testing for example, in particular to define instances of QuickCheck's Arbitrary.

These generators are called Boltzmann samplers, as introduced by Duchon et al. (2004). They produce finite values of a given type and about a given size (the number of constructors) in linear time. And the distribution is uniform when conditioned to a fixed size: two values with the same size occur with the same probability.

Future work

The implementation is currently focused on Haskell datatypes, even though the theory of Boltzmann samplers is much more general than that. I plan to generalize the code to make it reusable in other frameworks and applications, as well as to provide better customizability.

I may also look for solutions to handle types that do not fit Boltzmann models.

The user experience is of course important, and I will be thinking about ways to present and document the library that can be understood by people who are not combinatorics wizards.

Further reading

- Boltzmann Samplers for the Random Generation of Combinatorial Structures P. Duchon, P. Flajolet, G. Louchard, G. Schaeffer.
  http://algo.inria.fr/flajolet/Publications/DuFlLoSc04.pdf
- http://hackage.haskell.org/package/generic-random

6.4.6 Automatic type inference from JSON

Report by:	Michal J. Gajda
Status:	stable

This rapid software development tool `json-autotype` interprets JSON data and converts them into Haskell module with data type declarations.

```
$ json-autotype input.json -o JSONTypes.hs
```

The generated declarations use automatically derived Aeson class instances to read and write data directly from/to JSON strings, and facilitate interaction with growing number of large JSON APIs.

Generated parser can be immediately tested on an input data:

```
$ runghc JSONTypes.hs input.json
```

The software can be installed directly from Hackage.

It uses sophisticated union type unification, and robustly interprets most ambiguities using clever typing.

The tool has reached maturity this year, and thanks to automated testing procedures it seems to robustly infer types for all JSON inputs considered valid by Aeson.

The author welcomes comments and suggestions at ⟨mjgajda@gmail.com⟩.

Further reading

http://hackage.haskell.org/packages/json-autotype

6.4.7 Exference

Report by:	Lennart Spitzner
Status:	experimental, active development

Exference is a tool aimed at supporting developers writing Haskell code by generating expressions from a type, e.g.

Input:

```
(Show b) => (a -> b) -> [a] -> [String]
```

Output:

```
\ f1 -> fmap (show . f1)
```

Input:

```
   (Monad m, Monad n)
=> ([a] -> b -> c) -> m [n a] -> m (n b)
-> m (n c)
```

Output:

```
\ f1 -> liftA2 (\ hs i ->
  liftA2 (\ n os -> f1 os n) i (sequenceA hs))
```

The algorithm does a proof search specialized to the Haskell type system. In contrast to Djinn, the well known tool with the same general purpose, Exference supports a larger subset of the Haskell type system - most prominently type classes. The cost of this feature is that Exference makes no promise regarding termination (because the problem becomes an undecidable one; a draft of a proof can be found in the pdf below). Of course the implementation applies a time-out.

There are two primary use-cases for Exference:

- In combination with typed holes: The programmer can insert typed holes into the source code, retrieve the expected type from ghc and forward this type to Exference. If a solution, i.e. an expression, is found and if it has the right semantics, it can be used to fill the typed hole.

- As a type-class-aware search engine. For example, Exference is able to answer queries such as Int → Float, where the common search engines like hoogle or hayoo are not of much use.

Since the last HCAR, development has slowed down but continued. Additions include minor optimizations, support for type declarations, improvements to the interface (simplifications of the expression, etc.) and expansion of the default environment.

Try it out by on IRC(freenode): exferenceBot is in #haskell and #exference.

Further reading

- https://github.com/lspitzner/exference
- https://github.com/lspitzner/exference/raw/master/exference.pdf

6.4.8 Lentil

Report by:	Francesco Ariis
Status:	working

Lentil helps the programmers who litter their code with `TODO`s and `FIXME`s.

Lentil goes through a project and outputs all issues in a pretty format, referencing their file/line position. As today it recognises Haskell, Javascript, C/C++, Python, Ruby, Pascal, Perl, Shell and Nix source files, plus plain .txt files.

Lentil syntax allows you to put `[tag]`s in your issues, which can then be used to filter/extract/export data.

Current version is 0.1.12.0, which introduces new flag-words, recognised languages (html, elm, coffeescript, typescript) and export formats (xml).

Further reading

- manual: http://ariis.it/static/articles/lentil/page.html
- decentralised issue tracking: http://ariis.it/static/articles/decentralised-lentil/page.html

6.4.9 Hoed – The Lightweight Algorithmic Debugger for Haskell

Report by:	Maarten Faddegon
Status:	active

Hoed is a lightweight algorithmic debugger that is practical to use for real-world programs because it works with any Haskell run-time system and does not require trusted libraries to be transformed.

To locate a defect with Hoed you annotate suspected functions and compile as usual. Then you run your program, information about the annotated functions is collected. Finally you connect to a debugging session using a webbrowser.

Using Hoed

Let us consider the following program, a defective implementation of a parity function with a test property.

```
isOdd :: Int -> Bool
isOdd n = isEven (plusOne n)

isEven :: Int -> Bool
isEven n = mod2 n == 0

plusOne :: Int -> Int
plusOne n = n + 1

mod2 :: Int -> Int
mod2 n = div n 2

prop_isOdd :: Int -> Bool
prop_isOdd x = isOdd (2*x+1)

main :: IO ()
main = printO (prop_isOdd 1)

main :: IO ()
main = quickcheck prop_isOdd
```

Using the property-based test tool QuickCheck we find the counter example 1 for our property.

```
./MyProgram
*** Failed! Falsifiable (after 1 test): 1
```

Hoed can help us determine which function is defective. We annotate the functions *isOdd*, *isEven*, *plusOne* and *mod2* as follows:

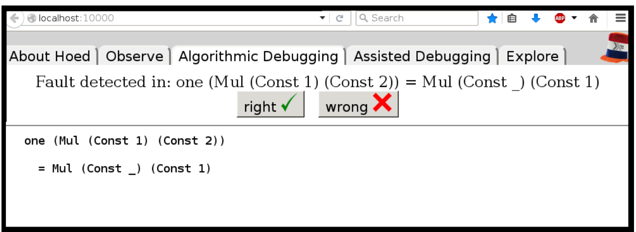
```
import Debug.Hoed.Pure

isOdd :: Int -> Bool
isOdd = observe "isOdd" isOdd'
isOdd' n = isEven (plusOne n)

isEven :: Int -> Bool
isEven = observe "isEven" isEven'
isEven' n = mod2 n == 0

plusOne :: Int -> Int
plusOne = observe "plusOne" plusOne'
plusOne' n = n + 1

mod2 :: Int -> Int
mod2 = observe "mod2" mod2'
mod2' n = div n 2

prop_isOdd :: Int -> Bool
prop_isOdd x = isOdd (2*x+1)

main :: IO ()
main = printO (prop_isOdd 1)
```

And run our program:

```
./MyProgram
False
Listening on http://127.0.0.1:10000/
```

Now you can use your webbrowser to interact with Hoed.

There is a classic algorithmic debugging interface in which you are shown computation statements, these are function applications and their result, and are asked to judge if these are correct. After judging enough computation statements the algorithmic debugger tells you where the defect is in your code.



In the explore mode, you can also freely browse the tree of computation statements to get a better understanding of your program. The observe mode is inspired by HOOD and gives a list of computation statements. Using regular expressions this list can be searched. Algorithmic debugging normally starts at the top of the tree, e.g. the application of `isOdd` to `(2*x+1)` in the program above, using explore or observe mode a different starting point can be chosen.

To reduce the number of questions the programmer has to answer, we added a new mode Assisted Algorithmic Debugging in version 0.3.5 of Hoed. In this mode (QuickCheck) properties already present in program code for property-based testing can be used to automatically judge computation statements

Further reading

- http://wiki.haskell.org/Hoed
- http://hackage.haskell.org/package/Hoed

6.4.10 Déjà Fu: Concurrency Testing

Report by:	Michael Walker
Status:	actively developed

Déjà Fu is a concurrency testing tool for Haskell. It provides a typeclass abstraction over a large subset of the functionality in the *Control.Concurrent* module hierarchy, and makes use of testing techniques pioneered in the imperative and object-oriented worlds.

The testing trades completeness for speed, by bounding the number of preemptions and yields in a single execution, as well as the overall length. This also allows testing of potentially non-terminating programs. All of these bounds are optional, however, and can be disabled, or changed.

A brief list of supported functionality:

- Threads: the `forkIO*` and `forkOn*` functions, although bound threads are not supported.
- Getting and setting capablities (testing default is two).
- Yielding and delaying.
- Mutable state: STM, `MVar`, and `IORef`.
- Relaxed memory for `IORef` operations: total store order (the testing default) and partial store order.
- Atomic compare-and-swap for `IORef`.
- Exceptions.
- All of the data structures in *Control.Concurrent.** and *Control.Concurrent.STM.** have typeclass-abstracted equivalents.

This is quite a rich set of functionality, although it is not complete. If there is something else you need, file an issue!

A new release, dejafu-0.3, has recently been pushed to Hackage. This fixes a number of bugs and greatly improves the performance. The main remaining inefficiency is the handling of asynchronous exceptions, which is likely to be a focus of work for the next release. No large API changes are anticipated, however.

Further reading

- http://hackage.haskell.org/package/dejafu
- The 2015 Haskell Symposium paper is available at http://bit.ly/1N2Lkw4; and a more up-to-date technical report is available at http://bit.ly/1SMHx4U.
- There are a number of blog posts on the functionality and implementation at https://www.barrucadu.co.uk.

6.4.11 The Remote Monad Design Pattern

Report by:	Andrew Gill
Participants:	Justin Dawson, Mark Grebe, James Stanton, David Young
Status:	active

The **remote monad design pattern** is a way of making Remote Procedure Calls (RPCs), and other calls that leave the Haskell eco-system, considerably less expensive. The idea is that, rather than directly call a remote procedure, we instead give the remote procedure call a service-specific monadic type, and invoke the remote procedure call using a monadic "send" function. Specifically, a **remote monad** is a monad that has its evaluation function in a remote location, outside the local runtime system.

By factoring the RPC into sending invocation and service name, we can group together procedure calls, and amortize the cost of the remote call. To give an example, Blank Canvas, our library for remotely accessing the JavaScript HTML5 Canvas, has a `send` function, `lineWidth` and `strokeStyle` services, and our remote monad is called `Canvas`:

```
send        :: Device -> Canvas a -> IO a
lineWidth   :: Double             -> Canvas ()
strokeStyle :: Text               -> Canvas ()
```

If we wanted to change the (remote) line width, the `lineWidth` RPC can be invoked by combining `send` and `lineWidth`:

```
send device (lineWidth 10)
```

Likewise, if we wanted to change the (remote) stroke color, the `strokeStyle` RPC can be invoked by combining `send` and `strokeStyle`:

```
send device (strokeStyle "red")
```

The key idea is that remote monadic commands can be locally combined before sending them to a remote server. For example:

```
send device (lineWidth 10 >> strokeStyle "red")
```

The complication is that, in general, monadic commands can return a result, which may be used by subsequent commands. For example, if we add a monadic command that returns a Boolean,

```
isPointInPath :: (Double,Double) -> Canvas Bool
```

we could use the result as follows:

```
send device $ do
   inside <- isPointInPath (0,0)
   lineWidth (if inside then 10 else 2)
   ...
```

The invocation of `send` can also return a value:

```
do res <- send device (isPointInPath (0,0))
   ...
```

Thus, while the monadic commands inside `send` are executed in a remote location, the results of those executions need to be made available for use locally.

We had a paper in the 2015 Haskell Symposium that discusses these ideas in more detail, and more recently, we have improved the packet mechanism to include an analog of the applicate monad structure, allowing for even better bundling. We have also improved the error handling capabilities. These ideas are implemented up in the hackage package remote-monad, which captures the pattern, and automatically bundled the monadic requests.

Further reading

http://ku-fpg.github.io/practice/remotemonad

7 Libraries, Applications, Projects

7.1 Language Features

7.1.1 Conduit

Report by:	Michael Snoyman
Status:	stable

While lazy I/O has served the Haskell community well for many purposes in the past, it is not a panacea. The inherent non-determinism with regard to resource management can cause problems in such situations as file serving from a high traffic web server, where the bottleneck is the number of file descriptors available to a process.

The left fold enumerator was one of the first approaches to dealing with streaming data without using lazy I/O. While it is certainly a workable solution, it requires a certain inversion of control to be applied to code. Additionally, many people have found the concept daunting. Most importantly for our purposes, certain kinds of operations, such as interleaving data sources and sinks, are prohibitively difficult under that model.

The conduit package was designed as an alternate approach to the same problem. The root of our simplification is removing one of the constraints in the enumerator approach. In order to guarantee proper resource finalization, the data source must always maintain the flow of execution in a program. This can lead to confusing code in many cases. In conduit, we separate out guaranteed resource finalization as its own component, namely the ResourceT transformer.

Once this transformation is in place, data producers, consumers, and transformers (known as Sources, Sinks, and Conduits, respectively) can each maintain control of their own execution, and pass off control via coroutines. The user need not deal directly with any of this low-level plumbing; a simple monadic interface (inspired greatly by the pipes package) is sufficient for almost all use cases.

Since its initial release, conduit has been through many design iterations, all the while keeping to its initial core principles. The conduit API has remained stable on version 1.2, which includes a lot of work around performance optimizations, including a stream fusion implementation to allow much more optimized runs for some forms of pipelines, and the codensity transform to provide better behavior of monadic bind.

Additionally, much work has gone into `conduit-combinators` and `streaming-commons`. The former provides a "batteries included" approach to conduit, containing a wide array of common functionality for both chunked data (like ByteString, Text, and Vector) and unchunked data. The latter contains common functionality useful to most streaming data frameworks, made available so that other libraries in this solution space can share a common code base.

There is a rich ecosystem of libraries available to be used with conduit, including cryptography, network communications, serialization, XML processing, and more.

Many conduit libraries are available via Hackage, Stackage Nightly, and LTS Haskell (just search for the word conduit). The main repository includes a tutorial on using the package.

Further reading

- https://www.stackage.org/package/conduit
- https://github.com/snoyberg/conduit#readme
- http://hackage.haskell.org/packages/archive/pkg-list.html#cat:conduit

7.1.2 GHC type-checker plugin for kind Nat

Report by:	Christiaan Baaij
Status:	actively developed

As of GHC version 7.10, GHC's type checking and inference mechanisms can be enriched by plugins. This particular plugin enriches GHC's knowledge of arithmetic on the type-level. Specifically it allows the compiler to reason about *equalities* of types of kind `GHC.TypeLits.Nat`.

GHC's type-checker's knowledge of arithmetic is virtually non-existent: it doesn't know addition is associative and commutative, that multiplication distributes over addition, etc. In a dependently-typed language, or in Haskell using singleton types, one can provide proofs for these properties and use them to type-check programs that depend on these properties in order to be (type-)correct. However, most of these properties of arithmetic over natural number are elementary school level knowledge, and it is cumbersome and tiresome to keep on providing and proving them manually. This type-checker plugin adds the knowledge of these properties to GHC's type-checker.

For example, using this plugin, GHC now knows that:

```
(x + 2)^(y + 2)
```

is equal to:

```
4*x*(2 + x)^y + 4*(2 + x)^y + (2 + x)^y*x^2
```

The way that the plugin works, is that it normalises arithmetic expressions to a normal form that very much resembles *Cantor normal form for ordinals*(http://en.wikipedia.org/wiki/Ordinal_arithmetic#Cantor_normal_form). Subsequently, it perform a simple syntactic equality of the two expressions. Indeed, in the example above, the latter expression is the normal form of the former expression.

The main test suite for the plugin can be found at: https://github.com/christiaanb/ghc-typelits-natnormalise/blob/master/tests/Tests.hs. It demonstrates what kind of *correct* code can be written without type equality annotations, or the use of `unsafeCoerce`.

One important aspect of this plugin is that it only enriches the type checker's knowledge of equalities, but not *in*equalities. That is, it does not allow GHC to solve constraints such as:

```
CmpNat (x + 2) (x + 3) ~ 'LT
```

The plugin is available on hackage, for GHC version 7.10 and higher:

```
$ cabal update
$ cabal install ghc-typelits-natnormalise
```

What's new since last HCAR:

- Support for interacting with other type-checker plugins, the first being http://hackage.haskell.org/package/ghc-typelits-extra.
- Prove more equalities (http://hackage.haskell.org/package/ghc-typelits-natnormalise-0.3.2/changelog).

Development focus for the plugin is on: proving more equalities, further testing, and improving its test suite.

Further reading

- http://hackage.haskell.org/package/ghc-typelits-natnormalise
- http://hackage.haskell.org/package/base/docs/GHC-TypeLits.html

7.1.3 Dependent Haskell

Report by: Richard Eisenberg
Status: work in progress

I am working on an ambitious update to GHC that will bring full dependent types to the language. In GHC 8, the Core language and type inference have already been updated according to the description in our ICFP'13 paper [1]. Accordingly, *all* type-level constructs are simultaneously kind-level constructs, as there is no distinction between types and kinds. Specifically, GADTs and type families are promotable to kinds. At this point, I conjecture that any construct writable in those other dependently-typed languages will be expressible in Haskell through the use of singletons.

After this phase, I will embark on working a proper Π-binder into the language, much along the lines of Adam Gundry's thesis on the topic [2]. Having Π would give us "proper" dependent types, and there would be no more need for singletons. A sampling of what I hope is possible when this work is done is online [3], excerpted here:

data Vec :: $*$ → Integer → $*$ **where**
  Nil :: Vec a 0
  (:::) :: a → Vec a n → Vec a (1 '+ n)
replicate :: π n. $\forall a.$ a → Vec a n
replicate @0 _ = Nil
replicate x = x ::: *replicate* x

Of course, the design here (especially for the proper dependent types) is preliminary, and input is encouraged.

Further reading

- [1]: *System FC with Explicit Kind Equality*, by Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. ICFP '13. http://www.cis.upenn.edu/~eir/papers/2013/fckinds/fckinds.pdf
- [2]: *Type Inference, Haskell and Dependent Types*, by Adam Gundry. PhD Thesis, 2013. https://personal.cis.strath.ac.uk/adam.gundry/thesis/
- [3]: https://github.com/goldfirere/nyc-hug-oct2014/blob/master/Tomorrow.hs
- Haskell Implementors' Workshop 2014 presentation on Dependent Haskell. Slides: http://www.cis.upenn.edu/~eir/talks/2014/hiw-dependent-haskell.pdf; Video: https://www.youtube.com/watch?v=O805YjOsQjI
- Repo for presentation on Dependent Haskell at the NYC Haskell Users' Group: https://github.com/goldfirere/nyc-hug-oct2014
- Wiki page with elements of the design: https://ghc.haskell.org/trac/ghc/wiki/DependentHaskell

7.1.4 Yampa

Report by: Ivan Perez

Yampa (Github: http://git.io/vTvxQ, Hackage: http://goo.gl/JGwycF), is a Functional Reactive Programming implementation in the form of a EDSL to define *Signal Functions*, that is, transformations of input signals into output signals (aka. *behaviours* in other FRP dialects).

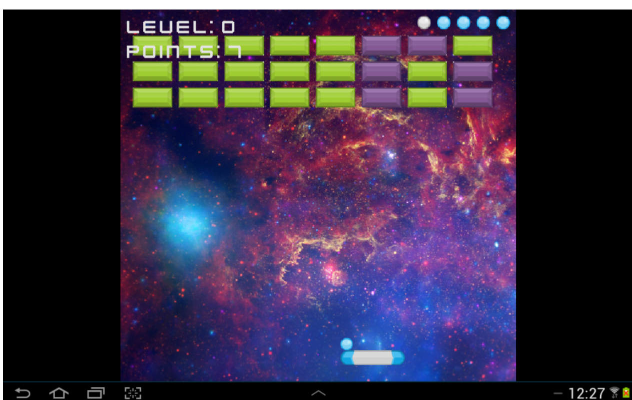
Yampa systems are defined as combinations of Signal Functions. The core of Yampa includes combinators to create constant signals, apply pointwise (or time-wise) functions to signals, access the running time of a signal

function, introduce delays and create loopbacks (carrying present output as future input). These systems can also be dynamic: their structure can change by using *switching* combinators, which enable the application of a different signal function at some point in the execution. Combined with combinators to deal with signal function collections, this enables a form of dynamic FRP in which new signals can be introduced, frozen, unfrozen, removed and altered at will.

Yampa is designed to guarantee *causality*: the value of an output signal at a time t can only depend on values of input signals at times $[0, t]$. Yampa restricts access to other signals only to the immediate past, by letting signals functions carry *state* for the future. FRP signal functions implement the Arrow and ArrowLoop typeclasses, making it possible to use both the arrow notation and arrow combinators. A suitable thinking model for FRP in Yampa is that of signal processing, in which components (signal functions) transform signals based on their present value and the component's internal state. Components can be serialized, applied in parallel, etc.

Unlike other implementations of FRP, Yampa enforces a strict separation of effects and pure transformations. All IO code must exist outside the Signal Functions, making Yampa systems easier to reason about and debug.

Yampa has been used to create both free/open-source and commercial games. Examples of the former include Frag (http://goo.gl/8bfSmz), a basic reimplementation of the Quake III Arena engine in Haskell, and Haskanoid (http://git.io/v8eq3), an arkanoid game featuring SDL graphics and sound with Wiimote & Kinect support. Examples of the latter include Keera Studios' Magic Cookies! (https://goo.gl/0A8z6i), a board game for Android written in Haskell and available via Google Play for Android store.



Yampa is actively maitained. The last updates have focused on introducing documentation, structuring the code, eliminating legacy code superceeded by other Haskell libraries, and increasing code quality in general. Over the years, performance in FRP has been an active topic of discussion and Yampa has been heavily optimised (games like Haskanoid have been clocked at over 700 frames per second on a standard PC). Also, because Yampa is *pure*, the introduction of parallelism is straightforward. In future versions, the benchmarking package `criterion` will be used to evaluate and increase performance. We encourage all Haskellers to participate by opening issues on our Github page (http://git.io/vTvxQ), adding improvements, creating tutorials and examples, and using Yampa in their next amazing Haskell games.

Extensions to Arrowized Functional Reactive Programming are an active research topic. The Functional Programming Laboratory at the University of Nottingham is working on several extensions to make Yampa more general and modular, facilitate other uses cases, increase performance and work around existing limitations. To collaborate with our research on FRP, please contact Ivan Perez at and Henrik Nilsson at .

7.2 Education

7.2.1 Holmes, Plagiarism Detection for Haskell

Report by:	Jurriaan Hage
Participants:	Brian Vermeer, Gerben Verburg

Holmes is a tool for detecting plagiarism in Haskell programs. A prototype implementation was made by Brian Vermeer under supervision of Jurriaan Hage, in order to determine which heuristics work well. This implementation could deal only with Helium programs. We found that a token stream based comparison and Moss style fingerprinting work well enough, if you remove template code and dead code before the comparison. Since we compute the control flow graphs anyway, we decided to also keep some form of similarity checking of control-flow graphs (particularly, to be able to deal with certain refactorings).

In November 2010, Gerben Verburg started to reimplement Holmes keeping only the heuristics we figured were useful, basing that implementation on `haskell-src-exts`. A large scale empirical validation has been made, and the results are good. We have found quite a bit of plagiarism in a collection of about 2200 submissions, including a substantial number in which refactoring was used to mask the plagiarism. A paper has been written, which has been presented at CSERC'13, and should become available in the ACM Digital Library.

The tool will be made available through Hackage at some point, but before that happens it can already be obtained on request from Jurriaan Hage.

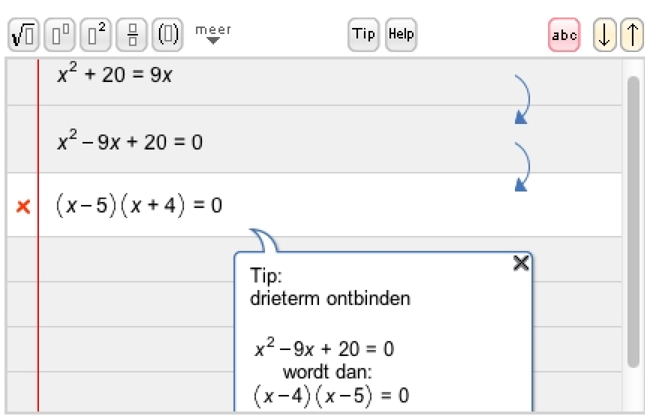
Contact

⟨J.Hage@uu.nl⟩

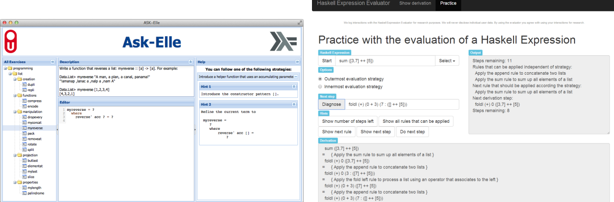
7.2.2 Interactive Domain Reasoners

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

Ideas (Interactive Domain-specific Exercise Assistants) is a joint research project between the Open University of the Netherlands and Utrecht University. The project's goal is to use software and compiler technology to build state-of-the-art components for intelligent tutoring systems (ITS) and learning environments. The 'ideas' software package provides a generic framework for constructing the expert knowledge module (also known as a domain reasoner) for an ITS or learning environment. Domain knowledge is offered as a set of feedback services that are used by external tools such as the digital mathematical environment (first/left screenshot), MathDox, and the Math-Bridge system. We have developed several domain reasoners based on this framework, including reasoners for mathematics, linear algebra, logic, learning Haskell (the Ask-Elle programming tutor) and evaluating Haskell expressions, and for practicing communication skills (the serious game Communicate!, second/right screenshot).



We have continued working on the domain reasoners that are used by our programming tutors. The Ask-Elle functional programming tutor lets you practice introductory functional programming exercises in Haskell. We have extended this tutor with QuickCheck properties for testing the correctness of student programs, and for the generation of counterexamples. We have analysed the usage of the tutor to find out how many student submissions are correctly diagnosed as right or wrong. Tim Olmer has developed a tutor in which a student can practice with evaluating Haskell expressions. Finally, Hieke Keuning has developed a programming tutor for imperative programming.



We are continuing our research in various directions. We are investigating feedback generation for axiomatic proofs for propositional logic, and are planning to add this to our logic tutor. We have just started on a statistics tutor. We also want to add student models to our framework and use these to make the tutors more adaptive, and develop authoring tools to simplify the creation of domain reasoners.

The library for developing domain reasoners with feedback services is available as a Cabal source package. In the near future we will update this package to work for ghc-7.10. We have written a tutorial on how to make your own domain reasoner with this library. We have also released our domain reasoner for mathematics and logic as a separate package.

Further reading

- Bastiaan Heeren, Johan Jeuring, and Alex Gerdes. Specifying Rewrite Strategies for Interactive Exercises. Mathematics in Computer Science, 3(3):349–370, 2010.
- Bastiaan Heeren and Johan Jeuring. Feedback services for stepwise exercises. Science of Computer Programming, Special Issue on Software Development Concerns in the e-Learning Domain, volume 88, 110–129, 2014.
- Tim Olmer, Bastiaan Heeren, Johan Jeuring. Evaluating Haskell expressions in a tutoring environment. Trends in Functional Programming in Education 2014.
- Hieke Keuning, Bastiaan Heeren, Johan Jeuring. Strategy-based feedback in a programming tutor. Computer Science Education Research Conference (CSERC 2014).
- Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and Thomas Binsbergen. Ask-Elle: an Adaptable

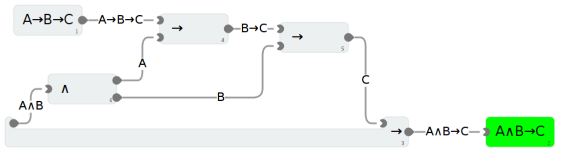
Programming Tutor for Haskell Giving Automated Feedback. Journal of Artificial Intelligence in Education 2016.

7.2.3 The Incredible Proof Machine

Report by:	Joachim Breitner
Status:	active development

The Incredible Proof Machine is a visual interactive theorem prover: Create proofs of theorems in propositional, predicate or other, custom defined logics simply by placing blocks on a canvas and connecting them. You can think of it as Simulink mangled by the Curry-Howard isomorphism.

It is also an addictive and puzzling game, I have been told.



The Incredible Proof Machine runs completely in your browser. While the UI is (unfortunately) boring standard JavaScript code with a spagetthi flavor, all the logical heavy lifting is done with Haskell, and compiled using GHCJS.

Further reading

- http://incredible.nomeata.de The Incredible Proof Machine
- https://github.com/nomeata/incredible Source Code
- http://www.joachim-breitner.de/blog/682-The_Incredible_Proof_Machine Announcement blog post

7.3 Mathematics, Numerical Packages and High Performance Computing

7.3.1 hblas

Report by:	Carter Tazio Schonwald
Participants:	Stephen Diehl and Csernik Flaviu Andrei
Status:	Actively Developed

`hblas` is high level, easy to extend BLAS/LAPACK FFI Binding for Haskell.

`hblas` has several attributes that *in aggregate* distinguish it from alternative BLAS/LAPACK bindings for Haskell.

1. Zero configuration install

2. FFI wrappers are written in Haskell

3. Provides the fully generality of each supported BLAS/LAPACK routine, in a type safe wrapper that still follows the naming conventions of BLAS and LAPACK.

4. Designed to be easy to extend with further bindings to BLAS/LAPACK routines (because there are many many specialized routines!)

5. Adaptively choses between unsafe vs safe foreign calls based upon estimated runtime of a computation, to ensure that long running `hblas` ffi calls interact safely with the GHC runtime and the rest of an application.

6. `hblas` is not an end user library, but is designed to easily interop with any array library that supports storable vectors.

Further reading

- http://www.wellposed.com
- http://www.github.com/wellposed/hblas
- http://hackage.haskell.org/package/hblas

7.3.2 Numerical

Report by:	Carter Tazio Schonwald
Status:	actively developed

The Numerical project, starting with the `numerical` package, has the goal of providing a general purpose numerical computing substrate for Haskell.

To start with, the `numerical` provides an extensible set of type classes suitable for both dense and sparse multi dimensional arrays, high level combinators for writing good locality code, and some basic matrix computation routines that work on both dense and sparse matrix formats.

The core Numerical packages, including `numerical`, are now in public pre-alpha as of mid May 2014, with on going active work as of November 2014.

Development of the numerical packages is public on github, and as they stabilize, alpha releases are being made available on hackage.

Further reading

- http://www.wellposed.com
- http://www.github.com/wellposed/numerical
- http://hackage.haskell.org/package/numerical

7.3.3 combinat

Report by:	Balázs Kőműves
Status:	actively developed

The `combinat` package is a broad-reaching combinatorics library. It provides functions to generate, manipulate, count and visualize various combinatorial objects, for example: trees, partitions, compositions, lattice paths, power series, permutations, braids, Young tableaux, and so on.

There is ASCII visualization for most structures, which makes it convenient to work in GHCi, and also `graphviz` and/or `diagrams` for some of them (the latter ones in a separate package).

Development is mostly done in short bursts, based mainly on the current (always changing) interests of the author.

Further reading

- http://hackage.haskell.org/package/combinat
- http://hackage.haskell.org/package/combinat-diagrams

7.3.4 petsc-hs

Report by:	Marco Zocca
Status:	experimental, actively developed

PETSc (http://www.mcs.anl.gov/petsc/) is an extensive C library for scientific computation. It provides a unified interface to distributed datastructures and algorithms for parallel solution of numerical problems, e.g. (non-)linear equation systems, time integration of dynamical systems, nonlinear (constrained) optimization. It is built upon MPI but abstracts it "out of sight"; however the API lets advanced users interleave computation and communication in order to experiment with resource usage and performance.

Many applications using PETSc are concerned with the solution of discretized PDEs for modelling physical phenomena, but the numerical primitives offered can be applied in many other contexts as well.

The aim of `petsc-hs` is to provide a compositional, type- and memory-safe way to interact with this library. The bindings are based on `inline-c` (https://hackage.haskell.org/package/inline-c) for quick experimentation with the C side.

Development of `petsc-hs` is public on github as of October 2015.

At present (November 2015), bindings for most of the basic functionality are available, memory pointers have been made lexically scoped and rudimentary exception handling is in place; the library is dynamically linked and can be tested with GHCi.

The immediate development plans are to move out of the experimental phase: currently the effort is concentrated on representing distributed mutable array operations and overall giving the library a more declarative interface while at the same time encapsulating the C version's best programming practices. Once this will be in place, a number of example PETSc programs will be provided and the API will be specialized to various use cases. Due to the multidisciplinary nature of this work, contributions, comments and test cases are more than welcome.

Further reading

https://github.com/ocramz/petsc-hs

7.4 Data Types and Data Structures

7.4.1 Transactional Trie

Report by:	Michael Schröder
Status:	stable

The transactional trie is a contention-free hash map for Software Transactional Memory (STM). It is based on the lock-free concurrent hash trie.

"Contention-free" means that it will never cause spurious conflicts between STM transactions operating on different elements of the map at the same time. Compared to simply putting a HashMap into a TVar, it is up to 8x faster and uses 10x less memory.

Further reading

- http://hackage.haskell.org/package/ttrie
- http://github.com/mcschroeder/thesis, in particular chapter 3, which includes a detailed discussion of the transactional trie's design and implementation, its limitations, and an evaluation of its performance.

7.4.2 fixplate

Report by:	Balázs Kőműves
Status:	experimental

The `fixplate` package is a re-implementation of Neil Mitchell's `uniplate` generic programming library, to work on data types realized as fixed points of functors (as opposed to plain recursive data types). It turns out that Functor, Foldable and Traversable instances are enough for this style of generic programming.

The original motivation for this exercise was the ability to add extra data to the nodes of an existing tree, motivated by attribute grammars. Recursion schemes also fit here very well, though they are less powerful.

Apart from the standard traversals, the library also provides a generic zipper, generic tries, generic tree hashing, a generic expression pretty-printer and generic tree visualization. The library itself is fully Haskell98-compatible, though some GHC extensions can make it more convenient to use.

Further reading

http://hackage.haskell.org/package/fixplate

7.4.3 generics-sop

Report by:	Andres Löh
Participants:	Andres Löh, Edsko de Vries

The `generics-sop` ("sop" is for "sum of products") package is a library for datatype-generic programming in Haskell, in the spirit of GHC's built-in `DeriveGeneric` construct and the `generic-deriving` package.

Datatypes are represented using a structurally isomorphic representation that can be used to define functions that work automatically for a large class of datatypes (comparisons, traversals, translations, and more). In contrast with the previously existing libraries, `generics-sop` does not use the full power of current GHC type system extensions to model datatypes as an n-ary sum (choice) between the constructors, and the arguments of each constructor as an n-ary product (sequence, i. e., heterogeneous lists). The library comes with several powerful combinators that work on n-ary sums and products, allowing to define generic functions in a very concise and compositional style.

The current release is 0.2.0.0.

A paper and a somewhat more recent, slightly longer, tutorial covering type-level programming as well as the use of this library, are available.

Further reading

- `generics-sop` package:
  https://hackage.haskell.org/package/generics-sop/
- Tutorial (summer school lecture notes):
  https://github.com/kosmikus/SSGEP/
- Paper:
  http://www.andres-loeh.de/TrueSumsOfProducts/

7.5 Databases and Related Tools

7.5.1 Persistent

Report by:	Greg Weber
Participants:	Michael Snoyman, Felipe Lessa
Status:	stable

The last HCAR announcement was for the release of Persistent 2.0, featuring a flexible primary key type.

Since then, persistent has mostly experienced bug fixes, including recent fixes and increased backend support for the new flexible primary key type.

Haskell has many different database bindings available, but most provide few usefeul static guarantees. Persistent uses knowledge of the data schema to provide a type-safe interface to the database. Persistent is designed to work across different databases, currently working on Sqlite, PostgreSQL, MongoDB, MySQL, Redis, and ZooKeeper.

Persistent provides a high-level query interface that works against all backends.

```
selectList [PersonFirstName ==. "Simon",
  PersonLastName ==. "Jones"] []
```

The result of this will be a list of Haskell records.

Persistent can also be used to write type-safe query libraries that are specific. esqueleto is a library for writing arbitrary SQL queries that is built on Persistent.

Future plans

Persistent is in a stable, feature complete state. Future plans are only to increase its ease the places where it can be easitly used:

- Declaring a schema separately from a record, possibly leveraging GHC's new annotations feature or another pattern

Persistent users may also be interested in Groundhog, a similar project.

Persistent is recommended to Yesod (→ 5.2.2) users. However, there is nothing particular to Yesod or even web development about it. You can have a type-safe, productive way to store data for any kind of Haskell project.

Further reading

- http://www.yesodweb.com/book/persistent
- http://hackage.haskell.org/package/esqueleto
- http://www.yesodweb.com/blog/2014/09/persistent-2
- http://www.yesodweb.com/blog/2014/08/announcing-persistent-2

7.5.2 Riak bindings

Report by:	Antonio Nikishaev
Status:	active development

riak is a Haskell binding to the Riak database. While stable and working, it has had only riak-1.∗ support. The author of this report entry has been recently working on fixing bugs and adding new riak-2.∗ features. Notable ones are: bucket types, high-level CRDT (Conflict-free replicated data types) support, basic search operations.

Further reading

- http://hackage.haskell.org/package/riak
- https://github.com/markhibberd/riak-haskell-client

7.5.3 Opaleye

Report by:	Tom Ellis
Status:	stable, active

Opaleye is an open-source library which provides an SQL-generating embedded domain specific language. It allows SQL queries to be written within Haskell in a typesafe and composable fashion, with clear semantics.

The project was publically released in December 2014. It is stable and actively maintained, and used in production in a number of commercial environments. Professional support is provided by Purely Agile.

Just like Haskell, Opaleye takes the principles of type safety, composability and semantics very seriously, and one aim for Opaleye is to be "the Haskell" of relational query languages.

In order to provide the best user experience and to avoid compatibility issues, Opaleye specifically targets PostgreSQL. It would be straightforward produce an adaptation of Opaleye targeting other popular SQL databases such as MySQL, SQL Server, Oracle and SQLite. Offers of collaboration on such projects would be most welcome.

Opaleye is inspired by theoretical work by David Spivak, and by practical work by the HaskellDB team. Indeed in many ways Opaleye can be seen as a spiritual successor to HaskellDB. Opaleye takes many ideas from the latter but is more flexible and has clearer semantics.

Further reading

http://hackage.haskell.org/package/opaleye

7.5.4 HLINQ - LINQ for Haskell

Report by:	Mantas Markevicius
Participants:	Mike Dodds, Jason Reich
Status:	Experimental

HLINQ is a Haskell implementation of the LINQ database query framework [1] modelled on Cheney *et al*'s T-LINQ system for F# [2]. Database queries in HLINQ are written in a syntax close to standard Haskell do notation:

```
getAge people = do                 getAge = [||do
p <- people                        p <- people db
guard ((name p) == "Edna")         guard ((name p) == "Edna")
return (age p)                     return (age p)||]
```

Queries can be composed using Template Haskell splicing operators, while type-safety rules provide additional correctness guarantees. Additionally, HLINQ is built on the HDBC library and uses prepared SQL statements protecting it against most SQL injection type attacks. Furthermore queries are avalanche-safe, meaning that for any query only a single SQL statement will be generated. Our system is in prototype stage, but microbenchmarks show performance competitive with HaskellDB.

The project is hosted on GitHub [3], with a technical report planned soon.

Further reading

1. Microsoft LINQ: https://msdn.microsoft.com/en-us/library/bb397926.aspx
2. Cheney, James, Sam Lindley, and Philip Wadler. "A practical theory of language-integrated query." ACM SIGPLAN Notices. Vol. 48. No. 9. ACM, 2013.
3. https://github.com/juventietis/HLINQ

7.5.5 YeshQL

Report by:	Tobias Dammers
Status:	Active, usable, somewhat stable

YeshQL is a library to bridge the Haskell / SQL gap by implementing a quasi-quoter that allows programmers to write SQL queries in plain SQL, adding metainformation as structured SQL comments. The latter allows the quasi-quoter to generate a type-safe API for these queries. YeshQL uses HDBC for the database backends, but doesn't depend on any particular HDBC driver.

The approach was stolen from the YesQL library for Clojure, and adapted to be more idiomatic in Haskell.

An example code snippet might look like this:

```
withTransaction db $ \conn -> do
    pageID:_ <- [yesh1|
        -- :: (Integer)
        -- :title:Text
        -- :body:Text
        INSERT INTO pages (title, body)
        VALUES (:title, :body)
        RETURNING id
        |]
        conn title body
    [yesh1|
        -- :: Integer
        INSERT
        INTO page_owners (page_id, owner_id)
        VALUES (:pageID, :userID)
        |]
        conn pageID currentUserID
    return pageID
```

YeshQL is somewhat production ready; I have used it on several real-world projects, with good success. However, it is still a bit rough around some edges, to note:

- While results are type safe, YeshQL can currently only generate query functions that return (lists of) tuples (for SELECT queries), row counts (UPDATE, INSERT, DELETE), or row IDs (INSERT). I would like to extend it such that the query functions can automatically convert entire rows to typed values other than tuples.

- A way of marking queries as "intended to return only one row" is currently missing. Such a feature would change the return type of a SELECT query from a list to a Maybe, or throw an exception when no row was found.
- Parser errors could use some love, and the parser could be made more robust overall.

All that said, contributions of any kind are more than welcome.

Further reading

- https://bitbucket.org/tdammers/yeshql
- http://hackage.haskell.org/package/yeshql
- https://github.com/krisajenkins/yesql (not my work)

7.6 User Interfaces

7.6.1 HsQML

Report by:	Robin KAY
Status:	active development

HsQML provides access to a modern graphical user interface toolkit by way of a binding to the cross-platform Qt Quick framework.

The library focuses on mechanisms for marshalling data between Haskell and Qt's domain-specific QML language. The intention is that QML, which incorporates both a declarative syntax and JavaScript code, can be used to design and animate the front-end of an application while being able to easily interface with Haskell code for functionality.

Status The latest version at time of press is `0.3.3.0`. Changes released since the previous edition of this report include support for rendering custom OpenGL graphics onto QML elements, facilities for managing object life-cycles with weak references and finalisers, and a number of bug fixes. It has been tested on the major desktop platforms: Linux, Windows, and MacOS.

Further reading

http://www.gekkou.co.uk/software/hsqml/

7.6.2 threepenny-gui

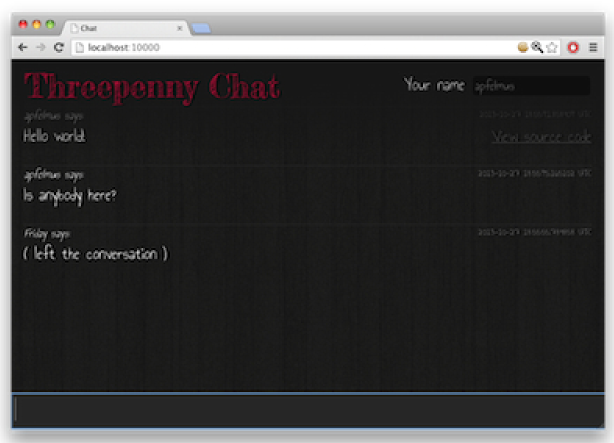
Report by:	Heinrich Apfelmus
Status:	active development

Threepenny-gui is a framework for writing graphical user interfaces (GUI) that uses the web browser as a display. Features include:

- *Easy installation.* Everyone has a reasonably modern web browser installed. Just install the library from Hackage and you are ready to go. The library is cross-platform.
- *HTML + JavaScript.* You have all capabilities of HTML at your disposal when creating user interfaces. This is a blessing, but it can also be a curse, so the library includes a few layout combinators to quickly create user interfaces without the need to deal with the mess that is CSS. A foreign function interface (FFI) allows you to execute JavaScript code in the browser.
- *Functional Reactive Programming (FRP)* promises to eliminate the spaghetti code that you usually get when using the traditional imperative style for programming user interactions. Threepenny has an FRP library built-in, but its use is completely optional. Employ FRP when it is convenient and fall back to the traditional style when you hit an impasse.

Status

The project is alive and kicking, the latest release is version `0.6.0.3`. You can download the library from Hackage and use it right away to write that cheap GUI you need for your project. Here a screenshot from the example code:



For a collection of real world applications that use the library, have a look at the gallery on the homepage.

Compared to the previous report, no major changes have been made. A bug related to garbage collection of event handlers has been fixed, and the library has been updated to work with the current Haskell ecosystem.

Current development

The library is still very much in flux, significant API changes are likely in future versions. The goal is to make GUI programming as simple as possible, and that just needs some experimentation.

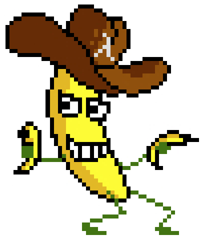
While Threepenny uses the web browser as a display, the goal was always to provide an environment for developing desktop applications. Recentely, a new platform for developing desktop applications with JavaScript has emerged, called Electron. I have successfully managed to connect Threepenny with the Electron platform, but I don't know how to best integrate this with the Haskell ecosystem, in particular Cabal. If you can offer any help with this, please let me know.

Further reading

- Project homepage: http://wiki.haskell.org/Threepenny-gui
- Example code: https://github.com/HeinrichApfelmus/threepenny-gui#examples
- Application gallery: http://wiki.haskell.org/Threepenny-gui#Gallery

7.6.3 reactive-banana

Report by:	Heinrich Apfelmus
Status:	active development



Reactive-banana is a library for functional reactive programming (FRP).

FRP offers an elegant and concise way to express interactive programs such as graphical user interfaces, animations, computer music or robot controllers. It promises to avoid the spaghetti code that is all too common in traditional approaches to GUI programming.

The goal of the library is to provide a solid foundation.

- Programmers interested in implementing FRP will have a *reference* for a *simple semantics* with a working implementation. The library stays close to the semantics pioneered by Conal Elliott.
- The library features an *efficient implementation*. No more spooky time leaks, predicting space & time usage should be straightforward.

The library is meant to be used in conjunction with existing libraries that are specific to your problem domain. For instance, you can hook it into any event-based GUI framework, like wxHaskell or Gtk2Hs. Several helper packages like reactive-banana-wx provide a small amount of glue code that can make life easier.

Status. With the release of version `1.0.0.0`, the development of the reactive-banana library has reached a milestone! I finally feel that the library does all the things that I wanted it to do.

In particular, compared to the previous report, the library now implements garbage collection for dynamically switched events. Also, the API no longer uses a phantom parameter to keep track of starting times; instead, a monadic approach is used. This simplifies the API for dynamic event switching, at the cost of requiring monadic types for some first-order combinators like `stepper`.

Additionally, there has been a small change concerning the semantics of the `Event` type: It is no longer possible to have multiple simultaneous occurrences in a single event stream. This forces the programmer to be more thoughtful about simultaneous event occurrences, a common source of bugs. The expressivity is the same, the old semantics can be recovered by using lists as occurrences.

Current development. With the library being complete, is there anything left to do? Well, of course, a library is never complete! However, my future focus will lie more on applications of FRP, rather than the implementation of the FRP primitives. For instance, I want to make more use of FRP in my `threepenny-gui` project, which is a library for writing graphical user interfaces in Haskell (→ 7.6.2). In turn, this will probably lead to improvements in the reactive-banana library, be it API revisions or performance tuning.

Further reading

- Project homepage: http://wiki.haskell.org/Reactive-banana
- Example code: http://wiki.haskell.org/Reactive-banana/Examples

7.6.4 fltkhs - GUI bindings to the FLTK library

Report by:	Aditya Siram
Status:	active

The `fltks` project is a set of bindings to the FLTK C++ toolkit (www.fltk.org). Coverage is fairly complete (85%) and it is easy to install and use. The main goal of this effort is to provide a low-cost, hassle-free way of creating self-contained, native GUI applications in pure Haskell that are portable to Windows, Linux and OSX.

FLTK was chosen because it is a mature toolkit and designed to be lightweight, portable and self-contained. In turn, `fltks` inherits these qualities with the additional benefit of having almost no dependencies outside of *base* and FLTK itself. This makes it very easy to get up and running with `fltks`.

`fltks` is also designed to be easy to use and learn. It tries to accomplish this by providing an API that matches the FLTK API as closely as possible so that a user can look up the pre-existing FLTK documentation for some function and in most cases be able to "guess" the corresponding Haskell function that delegates to it. Additionally `fltks` currently ships with 15 demos which are exact ports of demos shipped with the FLTK distribution so the user can study the code side-by-side. In most cases there is direct correspondence.

`fltks` is also extensible in a couple of ways. Firstly, the user can create custom GUI widgets in pure Haskell by simply overriding some key C++ functions with Haskell functions. Secondly, it is easy to add third-party widgets without touching the core bindings. Meaning if there is a useful FLTK widget that is not part of the FLTK distribution, the user can easily wrap it and publish it as a separate package without ever touching these bindings. Hopefully this fosters contribution allowing `fltks` to keep up with the FLTK ecosystem and even outpace it since users are now able to create new widgets in pure Haskell.

Ongoing work includes not only covering 100% of the API and porting all the demos but also adding support for FLUID (http://en.wikipedia.org/wiki/FLUID), the FLTK GUI builder. Haskellers will then be able to take any existing FLTK app which uses FLUID to build the user interface and migrate it to Haskell.

Contributions are welcome!

Further reading

https://hackage.haskell.org/package/fltkhs

7.6.5 wxHaskell

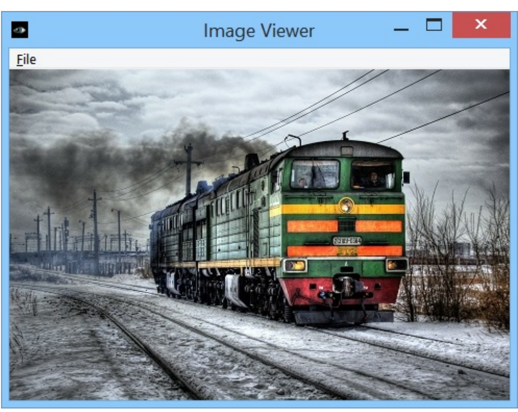
Report by:	Henk-Jan van Tuyl
Status:	active development

Since the previous HCAR, not much has changed, but there are plans to adapt wxHaskell to wxWidgets 3.1 and GHC 8.0 if necessary. New project participants are welcome.

wxHaskell is a portable and native GUI library for Haskell. The goal of the project is to provide an industrial strength GUI library for Haskell, but without the burden of developing (and maintaining) one ourselves.

wxHaskell is therefore built on top of wxWidgets: a comprehensive C++ library that is portable across all major GUI platforms; including GTK, Windows, X11, and MacOS X. Furthermore, it is a mature library (in development since 1992) that supports a wide range of widgets with the native look-and-feel.

A screen printout of a sample wxHaskell program:



Further reading

https://wiki.haskell.org/WxHaskell

7.7 Graphics and Audio

7.7.1 vect

Report by:	Balázs Kőműves
Status:	mostly stable

The `vect` package is low-dimensional linear algebra library intended specifically for computer graphics. It provides types and operations in 2, 3 and 4 dimensions, and is more-or-less feature-complete. OpenGL support is available as a separate package.

The library is intentionally monomorphic, providing as base fields the concrete types Float and Double. The monomorphicity makes life easier for both the user and the compiler, and we think that for graphics these two types cover most of the typical use cases. Nevertheless, a third, polymorphic version may be added in the future (until that happens, there is a polymorphic fork on Hackage).

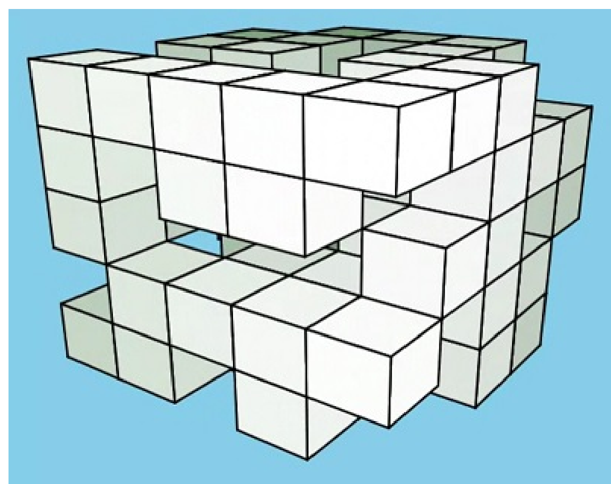
Further reading

- http://hackage.haskell.org/package/vect
- http://hackage.haskell.org/package/vect-opengl

7.7.2 diagrams

Report by:	Brent Yorgey
Participants:	many
Status:	active development

The diagrams framework provides an embedded domain-specific language for declarative drawing. The overall vision is for diagrams to become a viable alternative to DSLs like MetaPost or Asymptote, but with the advantages of being *declarative*—describing what to draw, not how to draw it—and *embedded*—putting the entire power of Haskell (and Hackage) at the service of diagram creation. There is always more to be done, but diagrams is already quite fully-featured, with a comprehensive user manual and a growing set of tutorials, a large collection of primitive shapes and attributes, many different modes of composition, paths, cubic splines, images, text, arbitrary monoidal annotations, named subdiagrams, and more.

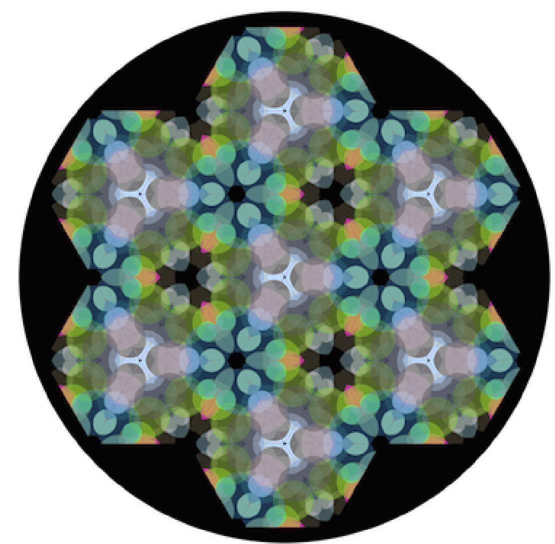


What's new

Diagrams development has currently stalled a bit, as most of the major contributors are busy with other things. However, this summer should bring a 1.4 release with some new features:

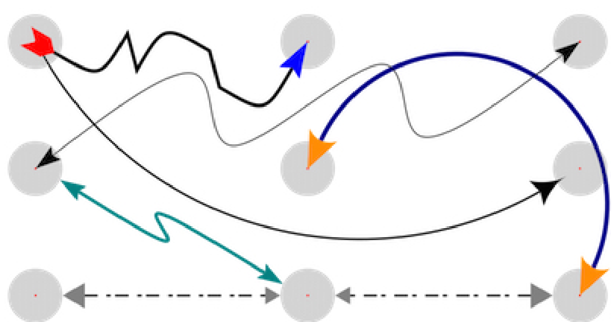
- B-spline support, and B-spline to cubic Bezier conversion
- Path union and intersection
- CSG support for 3D diagrams
- New techniques and tools for drawing 2D projections of 3D diagrams, illustrated above
- Constraint-based layout

We are also continuing work on using `stack` to create a system for easier, more reproducible builds, which will benefit both users and developers, and form the basis for much more comprehensive continuous integration testing.



Contributing

There is plenty of exciting work to be done; new contributors are welcome! Diagrams has developed an encouraging, responsive, and fun developer community, and makes for a great opportunity to learn and hack on some "real-world" Haskell code. Because of its size, generality, and enthusiastic embrace of advanced type system features, diagrams can be intimidating to would-be users and contributors; however, we are actively working on new documentation and resources to help combat this. For more information on ways to contribute and how to get started, see the Contributing page on the diagrams wiki: http://haskell.org/haskellwiki/Diagrams/Contributing, or come hang out in the `#diagrams` IRC channel on freenode.

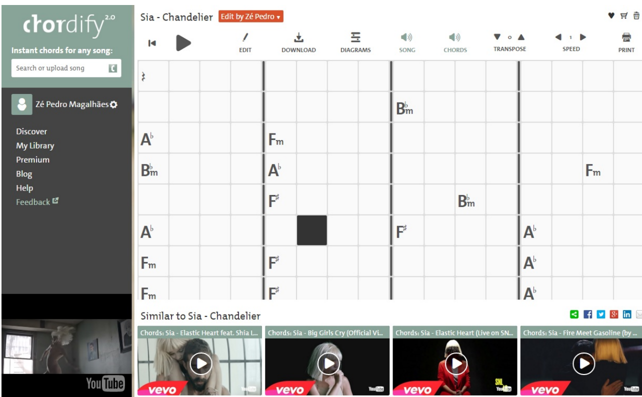


Further reading

- http://projects.haskell.org/diagrams
- http://projects.haskell.org/diagrams/gallery.html
- http://haskell.org/haskellwiki/Diagrams
- http://github.com/diagrams
- http://ozark.hendrix.edu/~yorgey/pub/monoid-pearl.pdf
- http://www.youtube.com/watch?v=X-8NCkD2vOw

7.7.3 Chordify

Report by:	Jeroen Bransen
Participants:	W. Bas de Haas, José Pedro Magalhães, Dion ten Heggeler, Tijmen Ruizendaal, Gijs Bekenkamp, Hendrik Vincent Koops
Status:	actively developed



Chordify is a music player that extracts chords from musical sources like Youtube, Deezer, Soundcloud, or your own files, and shows you which chord to play when. The aim of Chordify is to make state-of-the-art music technology accessible to a broader audience. Our interface is designed to be simple: everyone who can hold a musical instrument should be able to use it.

Behind the scenes, we use the sonic annotator for extraction of audio features. These features consist of the downbeat positions and the tonal content of a piece of music. Next, the Haskell program HarmTrace takes these features and computes the chords. HarmTrace uses a model of Western tonal harmony to aid in the chord selection. At beat positions where the audio matches a particular chord well, this chord is used in final transcription. However, in case there is uncertainty about the sounding chords at a specific position in the song, the HarmTrace harmony model will select the correct chords based on the rules of tonal harmony.

We have recently expanded our team and we are currently redesigning and expanding the Haskell backend. We want to use machine learning techniques to improve the chord extraction algorithm based on user edits. Furthermore, we are currently implementing a system that merges edits from various users into one single corrected version, of which preliminary results can be found here.

The code for HarmTrace is available on Hackage, and we have ICFP'11 and ISMIR'12 publications describing some of the technology behind Chordify.

Further reading

https://chordify.net

7.7.4 csound-expression

Report by:	Anton Kholomiov
Status:	active, experimental

The csound-expression is a Haskell framework for electronic music production. It's based on very efficient and feature rich software synthesizer Csound. The Csound is a programming language for music production. The csound-expression strives to be as simple and responsive as it can be. Features include support for almost all Csound audio units, composable GUIs, FRP for event scheduling, MIDI support and many others. Library contains many beautiful instruments (see the companion package csound-catalog). It can be used for real-time performances.

The csound-expression is a Csound code generator. The flexible nature of Csound (it's written in C and has wonderful API) allows to use the produced code on any desktop OS, Android, iOS, Raspberry Pi, Unity, within many other languages. We can create audio engines with Haskell.

With Csound it inherits many cutting edge sound design techniques like granular synthesis or hyper vectorial synthesis, ambisonics. It's based on Csound but there is no need to know Csound to create something interesting with it. You can just read the guide of the library and start coding your own synthesizers or electronic music: https://github.com/spell-music/csound-expression/blob/master/tutorial/Index.md.

So what's new? Let's review the most prominent features:

- The new release adds support for the famous Padsynth algorithm. It allows to create very pleasing pads and natural textures. Also there are some predefined instruments that explore the realms of padsynth (see the details at https://github.com/spell-music/csound-expression/blob/master/tutorial/chapters/Padsynth.md).

- The Micro-tonal tunings were added. User can play with custom tunings live and use them to compose the music. There are many predefined ancient and modern tunings available. You can discover the music of pre equal temperament era! You can listen the music how the Bach, Mozart and Chopin had listened to it (see the details at https://github.com/spell-music/csound-expression/blob/master/tutorial/chapters/Tuning.md).

- Support for Csound API was implemented. With Csound API you can use the Csound as a library and insert your Haskell generated Csound files into another programs. Many languages can use Csound with API (Python, Java, C, C++, Lua, and many more). you can check out the guide on how to

use the Haskell generated code inside Python here https://github.com/spell-music/csound-expression/blob/master/tutorial/chapters/CsoundAPI.md.

- The ping-pong delay was added to the stack of effects.

- New release adds support for monophonic synthesizers. They are very useful for modern electronic dance music.

- Functions for creation of FM-like synthesizers were added. You can construct a synthesizer as graph with FM units at the nodes. You can use as many units as you wish!

The library was used in real musical applications!

- The desktop synthesizer called tiny-synth was developed with Python and Haskell. The UI was written in Python with the help of wxPython library and the audio engine was written in Haskell. The synthesizer contains many instruments that were taken from the collection of instruments defined in the package csound-catalog (it's also available on Hackage). You can look at the source code of the project: https://github.com/spell-music/csound-expression/blob/master/tutorial/chapters/CsoundAPI.md. The synthesizer can be played live with MIDI-keyboard. It's designed to be simple to use. It contains about 100 instruments.

- The Haskell generated code was tested on stage in the real gig! It has happened on 3 of April. The Haskell program was used as a real-time synthesizer. The notes were triggered with MIDI-keyboard and the sound was produced with instruments created with Haskell. It was well received by the audience. The concert has lasted for an hour. You can hear the recordings of the music at soundcloud: https://soundcloud.com/kailash-project.

- I've made some new tracks that use the library. You can listen to them at my soundcloud page: https://soundcloud.com/anton-kho.

The future plans for the library is to improve documentation and create a musical Android application that is based on Haskell.

The library is available on Hackage. See the packages csound-expression, csound-sampler and csound-catalog.

Further reading

https://github.com/anton-k/csound-expression
http://csound.github.io/

7.7.5 hmidi

Report by:	Balázs Kőműves
Status:	stable

The `hmidi` package provides bindings to the OS-level MIDI services, allowing Haskell programs to communicate with physical or virtual MIDI devices, for example MIDI keyboards, controllers, synthesizers, or music software.

Supported operating systems are Mac OS X and Windows. Linux (ALSA) support may be added at some future time.

An example application is provided by the `launchpad-control` package, which provides a high-level interface to the Novation Launchpad MIDI controller.

Further reading

- http://hackage.haskell.org/package/hmidi
- http://hackage.haskell.org/package/launchpad-control

7.8 Text and Markup Languages

7.8.1 lhs2TEX

Report by:	Andres Löh
Status:	stable, maintained

This tool by Ralf Hinze and Andres Löh is a preprocessor that transforms literate Haskell or Agda 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.

The program is stable and can take on large documents.

The current version is 1.19 and has been released in April 2015. Development repository and bug tracker are on GitHub. The tool is mostly in plain maintenance mode, although there are still vague plans for a complete rewrite of lhs2TEX, hopefully cleaning up the internals and making the functionality of lhs2TEX available as a library.

Further reading

- http://www.andres-loeh.de/lhs2tex
- https://github.com/kosmikus/lhs2tex

7.8.2 pulp

Report by:	Daniel Wagner
Participants:	Daniel Wagner, Michael Greenberg
Status:	Not yet released

Anybody who has used LaTeX knows that it is a fantastic tool for typesetting; but its error reporting leaves much to be desired. Even simple documents that use a handful of packages can produce hundreds of lines of uninteresting output on a successful run. Picking out the parts that require action is a serious chore, and locating the right part of the document source to change can be tiresome when there are many files.

Pulp is a parser for LaTeX log files with a small but expressive configuration language for identifying which messages are of interest. A typical run of pulp after successfully building a document produces no output; this makes it very easy to spot when something has gone wrong. Next time you want to produce a great paper, process your log with pulp!

Features

- LaTeX log parser with special-case support for many popular packages and classes
- Expressive configuration language
  - Filter out document-specific unimportance
  - Increase verbosity as the document nears completion
- Uniform error reporting format with file and line information
- Instructions for use with latexmk
- Rudimentary Windows support

Further reading

http://github.com/dmwit/pulp

7.8.3 Unicode things

Report by:	Antonio Nikishaev
Status:	work in progress

Many programming languages offer non-existing or very poor support for Unicode. While many think that Haskell is not one of them, this is not completely true. The way-to-go library of Haskell's string type, Text, only provides codepoint-level operations. Just as a small and very elementary example: two "Haskell café" strings, first written with the 'é' character, and the second with the 'e' character followed by a combining acute accent character, obviously have a correspondence for many real-world situations. Yet they are entirely different and unconnected things for Text and its operations.

And even though there is `text-icu` library offering proper Unicode functions, it has a form of FFI bindings to C library (and that is painful, especially for Windows users). More so, its API is very low-level and incomplete.

`Prose` is a work-in-progress pure Haskell implementation of Unicode strings. Right now it's completely unoptimized. Implemented parts are normalization algorithms and segmentation by graphemes and words.

`Numerals` is pure Haskell implementation of CLDR (Common Language Data Repository, Unicode's locale data) numerals formatting.

Contributions and comments are always welcome!

Further reading

- http://lelf.lu/prose
- https://github.com/llelf/prose
- https://github.com/llelf/numerals

7.8.4 Ginger

Report by:	Tobias Dammers
Status:	Active, usable, not feature complete

Ginger is a Haskell implementation of the Jinja2 HTML template language. Unlike most existing Haskell templating solutions, Ginger expands templates at runtime, not compile time; this is a deliberate design decision, intended to support a typical rapid-cycle web development workflow. Also unlike most existing Haskell HTML DSLs, Ginger is completely unaware of the DOM, and does not enforce well-formed HTML. Just like Jinja2, however, it does distinguish HTML source and raw values at the type level, meaning that HTML encoding is automatic and (mostly) transparent, avoiding the most common source of XSS vulnerabilities. For a quick impression of what Ginger syntax looks like:

```
<section class="page">
    <h1>{{ page.title }}</h1>
    {% if page.image %}
        <img class="page-image"
             src="{{ page.image.thumbURL }}" />
    {% endif %}
    <section class="teaser">
      {{ page.teaser }}
    </section>
    <section class="content">
      {{ page.body|markdown }}
    </section>
    <section class="page-meta">
        Submitted by {{ page.author }} on
        {{ page.submitted|formatDate('%Y-%m-%d') }}
    </section>
</section>
```

All the important features of Jinja2 have been implemented, and the library is fully usable for production work. Some features of the original Jinja2 have been left out because the author considers them Pythonisms; others are missing simply because they haven't been

implemented yet. Other planned improvements include TemplateHaskell support (which would allow programmers to compile Ginger templates directly into the binary, and perform template compilation at compile time rather than runtime), a built-in caching mechanism, and more configuration options. I am also planning on overhauling the testing setup to use HUnit, QuickCheck, and Tasty, rather than (or in addition to) the current shell-script-based simulation tests. Contributions of any kind are very welcome.

In the near future, the most important things to add will be built-in filters to get closer to Jinja2 feature parity on that front.

Further reading

- https://bitbucket.org/tdammers/ginger
- http://hackage.haskell.org/package/ginger
- http://jinja2.pocoo.org (the original Ginger, not my work)

7.9 Natural Language Processing

7.9.1 NLP

Report by: Eric Kow

The Haskell Natural Language Processing community aims to make Haskell a more useful and more popular language for NLP. The community provides a mailing list, Wiki and hosting for source code repositories via the Haskell community server.

The Haskell NLP community was founded in March 2009. The list is still growing slowly as people grow increasingly interested in both natural language processing, and in Haskell.

At the present, the mailing list is mainly used to make announcements to the Haskell NLP community. We hope that we will continue to expand the list and expand our ways of making it useful to people potentially using Haskell in the NLP world.

New packages

- *Earley-0.8.0* (Olle Fredriksson)
  This (Text.Earley) is a library consisting of two parts:
  1. **Text.Earley.Grammar**: An embedded context-free grammar (CFG) domain-specific language (DSL) with semantic action specification in applicative style.

     An example of a typical expression grammar working on an input tokenized into strings is the following:

```
expr :: Grammar r String (Prod r String String Expr)
expr = mdo
  x1 ← rule $ Add <$> x1 <* namedSymbol "+" <*> x2
      <|> x2
      <?> "sum"
  x2 ← rule $ Mul <$> x2 <* namedSymbol "*" <*> x3
      <|> x3
      <?> "product"
  x3 ← rule $ Var <$> (satisfy ident <?> "identifier")
      <|> namedSymbol "(" *> x1 <* namedSymbol ")"
  return x1
  where
    ident (x: _) = isAlpha x
    ident _ = False
```

  2. **Text.Earley.Parser**: An implementation of (a modification of) the Earley parsing algorithm.

     To invoke the parser on the above grammar, run e.g. (here using words as a stupid tokeniser):

```
fullParses $ parser expr $ words "a + b * ( c + d )"
 = ([Add (Var "a") (Mul (Var "b")
(Add (Var "c") (Var "d")))]
  , Report {...}
  )
```

     Note that we get a list of all the possible parses (though in this case there is only one).

  https://github.com/ollef/Earley

Further reading

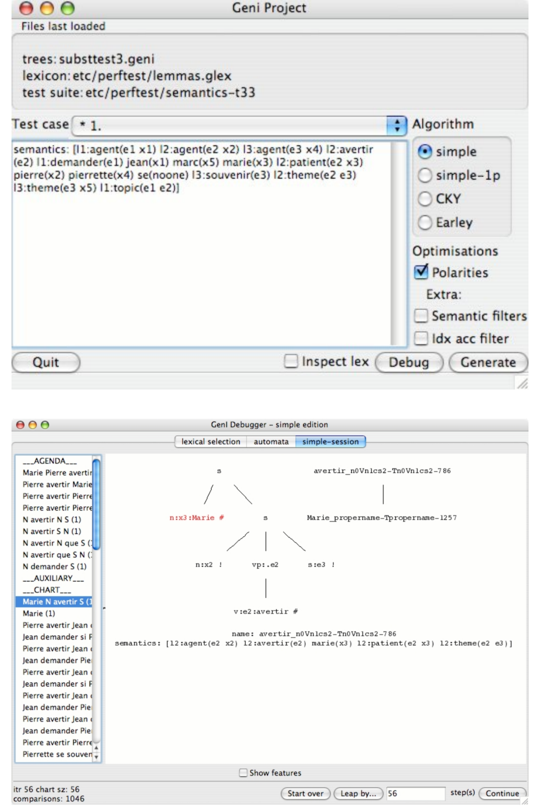
- The Haskell NLP page http://projects.haskell.org/nlp

7.9.2 GenI

Report by: Eric Kow

GenI is a surface realizer for Tree Adjoining Grammars. Surface realization can be seen a subtask of natural language generation (producing natural language utterances, e.g., English texts, out of abstract inputs). GenI in particular takes a Feature Based Lexicalized Tree Adjoining Grammar and an input semantics (a conjunction of first order terms), and produces the set of sentences associated with the input semantics by the grammar. It features a surface realization library, several optimizations, batch generation mode, and a graphical debugger written in wxHaskell. It was developed within the TALARIS project and is free software licensed under the GNU GPL, with dual-licensing available for commercial purposes.

GenI is now mirrored on GitHub, with its issue tracker and wiki and homepage also hosted there. The most recent release, GenI 0.24 (2013-09-18), allows for custom semantic inputs, making it simpler to use GenI in a wider variety for applications. This has recently been joined by a companion geni-util package which offers a rudimentary geniserver client and a reporting tool for grammar debugging.



GenI is available on Hackage, and can be installed via cabal-install, along with its GUI and HTTP server user interfaces. For more information, please contact us on the geni-users mailing list.

Further reading

- http://github.com/kowey/GenI
- http://projects.haskell.org/GenI
- Paper from Haskell Workshop 2006: http://hal.inria.fr/inria-00088787/en
- http://websympa.loria.fr/wwsympa/info/geni-users

7.10 Embedding DSLs for Low-Level Processing

7.10.1 CλaSH

Report by:	Christiaan Baaij
Participants:	Jan Kuper, Arjan Boeijink, Rinse Wester
Status:	actively developed

The first line of the package description on hackage is:

> CλaSH (pronounced 'clash') is a functional hardware description language that borrows its syntax and semantics from the functional programming language Haskell.

In essence, however, it is a combination of:

- A Haskell library containing data types and functions for circuit design: http://hackage.haskell.org/package/clash-prelude.
- A compiler that transforms the Haskell code to low-level synthesisable VHDL or SystemVerilog: http://hackage.haskell.org/package/clash-ghc.

Of course, the compiler cannot transform arbitrary Haskell code to hardware, but only the *structural* subset of Haskell. This subset is vaguely described as the *semantic* subset of Haskell from which a *finite* structure can be inferred, and hence excludes unbounded recursion. The CλaSH compiler is thus a proper compiler (based on static analysis), and *not* an embedded Domain Specific Language (DSL) such as Kansas Lava.

CλaSH has been in active development since 2010. Since then we have significantly improved stability, enlarged the subset of transformable Haskell, improved performance of the compiler, and added (System)Verilog generation. And, perhaps most importantly, vastly improved documentation.

CλaSH is available on Hackage, for GHC version 7.10 and higher:

```
$ cabal update
$ cabal install clash-ghc
```

What's new since last HCAR:

- CλaSH can now generate, next to VHDL-93 and SystemVerilog-2005, Verilog-2001.
- Support for memory primitives whose content can be initialised from a file.
- Major overhaul and extension of the `Vector` module. All functions in `Vector` are now synthesisable to VHDL/(System)Verilog.

Development plans for CλaSH are:

- Behavioural synthesis of unbounded recursion (by Ingmar te Raa).
- Use a dependently typed internal core language, so that we can use both Haskell/GHC and Idris http://http://www.idris-lang.org/ as *front-end* language for circuit design (by Christiaan Baaij).

Further reading

http://www.clash-lang.org

7.10.2 Feldspar

Report by:	Emil Axelsson
Status:	active development

Feldspar is a domain-specific language for digital signal processing (DSP). The language is embedded in Haskell and is currently being developed by projects at Chalmers University of Technology, SICS Swedish ICT AB and Ericsson AB.

The motivating application of Feldspar is telecoms processing, but the language is intended to be useful

for DSP and numeric code in general. The aim is to allow functions to be written in pure functional style in order to raise the abstraction level of the code and to enable more high-level optimizations. The current version consists of a library of numeric and array processing operations as well as a code generator producing C code for running on embedded targets.

The official packages feldspar-language and feldspar-compiler contain the language for pure computations and its C back end, respectively.

Additionally, we are working on a completely new implementation of Feldspar, RAW-Feldspar (not yet released, but fully usable). This implementation uses a slightly different language design that gives better control over things like memory allocation. It also extends Feldspar with a monad that supports interaction with the operating system, calls to external C libraries, concurrency, etc.

Ongoing work involves using RAW-Feldspar to implement more high-level libraries for streaming and interactive programs. Two examples of such libraries are:

- zeldspar – a Ziria-like EDSL
- feldspar-synch – a synchronous data-flow library

raw-feldspar-mcs is a library built on top of RAW-Feldspar that generates code for running on NUMA architectures such as the Parallella.

There is also ongoing work to generate VHDL from RAW-Feldspar programs.

Further reading

- Official home page: http://feldspar.github.io

7.11 Games

7.11.1 EtaMOO

Report by:	Rob Leslie
Status:	experimental, active development

EtaMOO is a new, experimental MOO server implementation written in Haskell. MOOs are network accessible, multi-user, programmable, interactive systems well suited to the construction of text-based adventure games, conferencing systems, and other collaborative software. The design of EtaMOO is modeled closely after LambdaMOO, perhaps the most widely used implementation of MOO to date.

Unlike LambdaMOO which is a single-threaded server, EtaMOO seeks to offer a fully multi-threaded environment, including concurrent execution of MOO tasks. To retain backward compatibility with the general MOO code expectation of single-threaded semantics, EtaMOO makes extensive use of software transactional memory (STM) to resolve possible conflicts among simultaneously running MOO tasks.

EtaMOO fully implements the MOO programming language as specified for the latest version of the LambdaMOO server, with the aim of offering drop-in compatibility. Several enhancements are also planned to be introduced over time, such as support for 64-bit MOO integers, Unicode MOO strings, and others.

Recent development has brought the project to a largely usable state. A major advancement was made by integrating the *vcache* library from Hackage for persistent storage — a pairing that worked especially well given EtaMOO's existing use of STM. Consequently, EtaMOO now has a native binary database backing with continuous checkpointing and instantaneous crash recovery. Furthermore, EtaMOO takes advantage of *vcache*'s automatic value cache with implicit structure sharing, so the entire MOO database need not be held in memory at once, and duplicate values (such as object properties) are stored only once in persistent storage.

Further development has incorporated optional support for the lightweight object WAIF data type as originally described and implemented for the LambdaMOO server. The *vcache* library was especially useful in implementing the persistent shared WAIF references for EtaMOO.

Future EtaMOO development will focus on feature parity with the LambdaMOO server, full Unicode support, and several additional novel features.

Latest development of EtaMOO can be seen on GitHub, with periodic releases also being made available through Hackage.

Further reading

- https://github.com/verement/etamoo
- https://hackage.haskell.org/package/EtaMOO
- https://en.wikipedia.org/wiki/MOO

7.11.2 scroll

Report by:	Joey Hess
Status:	stable, complete

Scroll is a roguelike game, developed in one week as an entry in the 2015 Seven Day Roguelike Challenge.

In scroll, you're a bookworm that's stuck on a scroll. You have to dodge between words and use spells to make your way down the page as the scroll is read. Go too slow and you'll get wound up in the scroll and crushed.

This was my first experience with using Haskell for game development, and I found it quite an interesting experience, and a great crutch in such an intense coding sprint. Strong typing and purely functional code saved me from many late night mistakes, until I eventually became so exhausted that String → String seemed like a good idea. Even infinite lists found a use; one of scroll's levels features a reversed infinite stream of consciousness based on Joyce's Ulysses. . .

Scroll was written in continuation passing style, and this turned out to be especially useful in developing its magic system, with spells that did things ranging from creating other spells, to using a quick continuation based threading system to handle background tasks, to letting the player enter the altered reality of a dream, from which they could wake up later.

I had a great time creating a game in such a short time with Haskell, and documenting my progress in 7 blog posts, and it's been well received by players.

Further reading

http://joeyh.name/code/scroll/

7.11.3 Nomyx

Report by:	Corentin Dupont
Status:	stable, actively developed

Nomyx is a unique game where you can change the rules of the game itself, while playing it! In fact, changing the rules is the goal of the game. Changing a rule is considered as a move. Of course even that can be changed! The players can submit new rules or modify existing ones, thus completely changing the behaviour of the game through time. The rules are managed and interpreted by the computer. They must be written in the Nomyx language, based on Haskell. This is the first complete implementation of a Nomic game on a computer.

At the beginning, the initial rules are describing:

- How to add new rules and change existing ones. For example a unanimity vote is necessary to have a new rule accepted.
- How to win the game. For example you win the game if you have 5 rules accepted.

But of course even that can be changed!

A first version has been released. A match is currently on-going, join us! A lot of learning material is available, including a video, a tutorial, a FAQ, a forum and API documentation.

If you like Nomyx, you can help! There is a development mailing list (check the website). The plan now is to create a new version were knowing haskell is not necessary to play.

Further reading

http://www.nomyx.net

7.11.4 Barbarossa

Report by:	Nicu Ionita
Status:	actively developed

Barbarossa is a UCI chess engine written completely in Haskell. UCI is one of 2 protocols used in the computer chess scene to communicate between a chess GUI and a chess engine. This way it is possible to write just the chess engine, which then works with any chess GUI.

I started in 2009 to write a chess engine under the name Abulafia. In 2012 I decided to rewrite the evaluation and search parts of the engine under the new name, Barbarossa.

My motivation was to demonstrate that even in a domain in which the raw speed of a program is very important, as it is in computer chess, it is possible to write competitive software with Haskell. The speed of Barbarossa (measured in searched nodes per second) is still far behind comparable engines written in C or C++. Nevertheless Barbarossa can compete with many engines - as it can be seen on the CCRL rating lists, where is it currently listed with a strength of about 2200 ELO.

Barbarossa uses a few techniques which are well known in the computer chess scene:

- in evaluation: material, king safety, piece mobility, pawn structures, tapped evaluation and a few other less important features
- in search: principal variation search, transposition table, null move pruning, killer moves, futility pruning, late move reduction, internal iterative deepening.

I still have a lot of ideas which could improve the strength of the engine, some of which address a higher speed of the calculations, and some, new chess related features, which may reduce the search tree.

The engine is open source and is published on github. The last released version is Barbarossa v0.3.0 from begin of October.

The version 0.4.0 is still in development and will be released probably end of May this year. Currently the version is about 100 ELO points stronger than the previously version (internal ratings).

Further reading

- https://github.com/nionita/Barbarossa/releases
- http://www.computerchess.org.uk/ccrl/404/

7.12 Others

7.12.1 ADPfusion

Report by:	Christian Höner zu Siederdissen
Status:	usable, active development

ADPfusion provides a low-level domain-specific language (DSL) for the formulation of dynamic programs with emphasis on computational biology and linguistics. We follow ideas established in algebraic dynamic programming (ADP) where a problem is separated into a grammar defining the search space and one or more algebras that score and select elements of the search space. The DSL has been designed with performance and a high level of abstraction in mind.

ADPfusion grammars are abstract over the type of terminal and syntactic symbols. Thus it is possible to use the same notation for problems over different input types. We directly support grammars over strings, sets (with boundaries, if necessary), and trees. Linear, context-free and multiple context-free languages are supported, where linear languages can be asymptotically more efficient both in time and space. ADPfusion is extendable by the user without having to modify the core library. This allows users of the library to support novel input types, as well as domain-specific index structures. The extension for tree-structured inputs is implemented in exactly this way and can serve as a guideline.

As an example, consider a grammar that recognizes palindromes. Given the non-terminal p, as well as parsers for single characters c and the empty input ϵ, the production rule for palindromes can be formulated as $p \rightarrow c\ p\ c \mid \epsilon$.

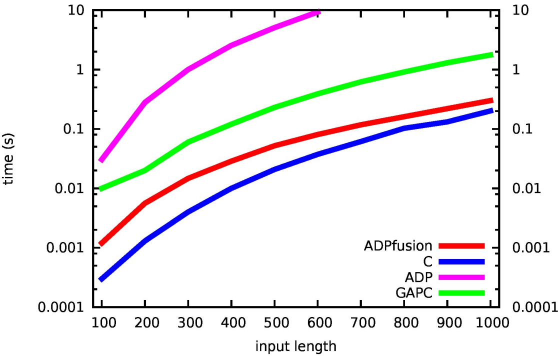
The corresponding ADPfusion code is similar:

```
p (f <<< c % p % c ||| g <<< e ... h)
```

We need a number of combinators as "glue" and additional evaluation functions f, g, and h. With $f\ c_1\ p\ c_2 = p\ \&\&\ (c_1 \equiv c_2)$ scoring a candidate, $g\ e\ = \texttt{True}$, and $h\ xs = \texttt{or}\ xs$ determining if the current substring is palindromic.

This effectively turns the grammar into a memo-function that then yields the optimal solution via a call to `axiom p`. Backtracking for co- and sub-optimal solutions is provided as well. The backtracking machinery is derived automatically and requires the user to only provide a set of pretty-printing evaluation functions.

As of now, code written in ADPfusion achieves performance close to hand-optimized `C`, and outperforms similar approaches (Haskell-based ADP, GAPC producing `C++`) thanks to stream fusion. The figure shows running times for the *Nussinov algorithm*.



The entry on generalized Algebraic Dynamic Programming (→ 7.12.2) provides information on the associated high-level environment for the development of dynamic programs.

Further reading

- http://www.bioinf.uni-leipzig.de/Software/gADP
- http://hackage.haskell.org/package/ADPfusion
- http://dx.doi.org/10.1145/2364527.2364559

7.12.2 Generalized Algebraic Dynamic Programming

Report by:	Christian Höner zu Siederdissen
Participants:	Sarah J. Berkemer
Status:	usable, active development

Generalized Algebraic Dynamic Programming (gADP) provides a solution for high-level dynamic programs. We treat the formal grammars underlying each DP algorithm as an algebraic object which allows us to *calculate* with them. gADP covers dynamic programming problems of various kinds: (i) we include linear, context-free, and multiple context-free languages (ii) over sequences, trees, and sets; and (iii) provide abstract algebras to combine grammars in novel ways.

Below, we describe the highlights our system offers in more detail:

Grammars Products

We have developed a theory of algebraic operations over linear and context-free grammars. This theory allows us to combine simple "atomic" grammars to create more complex ones.

With the compiler that accompanies our theory, we make it easy to experiment with grammars and their products. Atomic grammars are user-defined and the algebraic operations on the atomic grammars are embedded in a rigerous mathematical framework.

Our immediate applications are problems in computational biology and linguistics. In these domains, algorithms that combine structural features on individual inputs (or tapes) with an alignment or structure

between tapes are becoming more commonplace. Our theory will simplify building grammar-based applications by dealing with the intrinsic complexity of these algorithms.

We provide multiple types of output. LaTeX is available to those users who prefer to manually write the resulting grammars. Alternatively, Haskell modules can be created. TemplateHaskell and QuasiQuoting machinery is also available turning this framework into a fully usable embedded domain-specific language. The DSL or Haskell module use ADPfusion (→ 7.12.1) with multitape extensions, delivering “close-to-C” performance.

Set Grammars

Most dynamic programming frameworks we are aware of deal with problems over sequence data. There are, however, many dynamic programming solutions to problems that are inherently non-sequence like. Hamiltonian path problems, finding optimal paths through a graph while visiting each node, are a well-studied example.

We have extended our formal grammar library to deal with problems that can not be encoded via linear data types. This provides the user of our framework with two benefits, easy encoding of problems based on set-like inputs and construction of dynamic programming solutions. On a more general level, the extension of ADPfusion and the formal grammars library shows how to encode new classes of problems that are now gaining traction and are being studied.

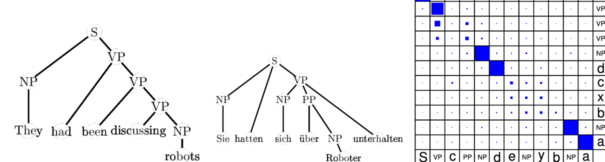
If, say, the user wants to calculate the shortest Hamiltonian path through all nodes of a graph, then the grammar for this problem is:

```
s (f <<< s % n ||| g <<< n ... h)
```

which states that a path s is either extended by a node n, or that a path is started by having just a first, single node n. Functions f and g evaluate the cost of moving to the new node. gADP has notions of sets with interfaces (here: for s) that provide the needed functionality for stating that all nodes in s have been visited with a final visited node from which an edge to n is to be taken.

Tree Grammars

Tree grammars are important for the analysis of structured data common in linguistics and bioinformatics. Consider two parse trees for english and german (from: Berkemer et al. *General Reforestation: Parsing Trees and Forests*) and the node matching probabilities we gain when trying to align the two trees:



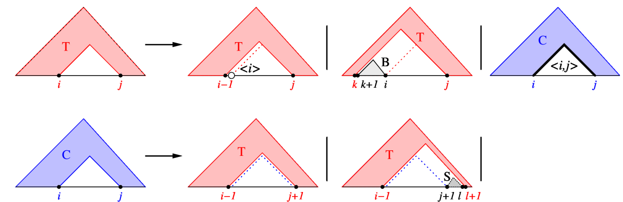
We can create the parse trees themselves with a normal context-free language on sequences. We can also compare the two sentences with, say, a Needleman-Wunsch style sequence alignment algorithm. However, this approach ignores the fact that parse trees encode grammatical structure inherent to languages. The comparison of sentences in english or german should be on the level of the structured parse tree, not the unstructured sequence of words.

Our extension of ADPfusion (→ 7.12.1) to forests as inputs allows us to deal with a variety of problems in complete analogy to sequence-based dynamic programming. This extension fully includes grammar products, and automatic outside grammars.

Automatic Outside Grammars

Our third contribution to high-level and efficient dynamic programming is the ability to automatically construct Outside algorithms given an Inside algorithm. The combination of an Inside algorithm and its corresponding Outside algorithm allow the developer to answer refined questions for the ensemble of all (sub-optimal) solutions.

The image below depicts one such automatically created grammar that parses a string from the Outside in. T and C are non-terminal symbols of the Outside grammar; the production rules also make use of the S and B non-terminals of the Inside version.



One can, for example, not only ask for the most efficient path through all cities on a map, but also answer which path between two cities is the most frequented one, given all possible travel routes. In networks, this allows one to determine paths that are chosen with high likelihood.

Multiple Context-Free Grammars

In both, linguistics and bioinformatics, a number of problems exist that can only be described with formal languages that are more powerful than context-free languages, but often have the form of two or more interleaved context-free languages (say: $a^n b^n c^n$). In RNA biology, pseudoknotted structures can be modelled in

this way, while in linguistics, we can model languages with crossing dependencies.

ADPfusion and the generalized Algebraic Dynamic Programming methodology have been extended to handle these kinds of grammars.

Further reading

- http://www.bioinf.uni-leipzig.de/Software/gADP/
- http://dx.doi.org/10.1109/TCBB.2014.2326155
- http://dx.doi.org/10.1007/978-3-319-12418-6_8

7.12.3 leapseconds-announced

Report by:	Björn Buckwalter
Status:	stable, maintained

The leapseconds-announced library provides an easy to use static LeapSecondTable with the leap seconds announced at library release time. It is intended as a quick-and-dirty leap second solution for one-off analyses concerned only with the past and present (i.e. up until the next as of yet unannounced leap second), or for applications which can afford to be recompiled against an updated library as often as every six months.

Version 2015 of leapseconds-announced contains all leap seconds up to 2015-07-01. A new version will be uploaded if/when the IERS announces a new leap second.

Further reading

https://hackage.haskell.org/package/leapseconds-announced

7.12.4 hledger

Report by:	Simon Michael
Status:	ongoing development; suitable for daily use

`hledger` is a cross-platform program (and Haskell library) for tracking money, time, or any other commodity, using double-entry accounting and a simple, editable text file format. `hledger` aims to be a reliable, practical tool for daily use, and provides command-line, curses-style, and web interfaces. It is a largely compatible Haskell reimplementation of John Wiegley's Ledger program. `hledger` is released under GNU GPLv3+.

`hledger`'s HCAR entry was last updated in the November 2011 report, but development has continued steadily, with 2-3 major releases each year.

Many new features and improvements have been introduced, making `hledger` much more useful. These include:

- Easier installation, using stack, system packages, or downloadable Windows binaries.
- A simpler and more robust web interface, with built-in help, balance charts, flexible transaction entry, and automatic browser startup
- A new curses-style interface, `hledger-ui`, is now included and fully supported
- The command-line interface is more robust, and is aware of terminal width, COLUMNS, and wide characters
- New commands: accounts, balancesheet, cashflow, incomestatement
- New add-on packages: ledger-autosync, `hledger-diff`, `hledger-interest`, and `hledger-irr`
- `hledger` can now report current value based on market prices (`-V`)
- The journal format has become richer, supporting more Ledger features such as balance assertions
- `hledger` journals and reports can be exported as CSV
- `hledger` now reads CSV files directly, using flexible conversion rules
- The balance command can show multiple columns, with per-period changes or ending balances
- Depth-limiting now interacts well with other features, making it effective for summarising
- `hledger-web`'s query language is richer and is also used by the command-line interface
- The Decimal library is used for representing amounts exactly
- Unicode is handled correctly
- Many commands are faster

Project updates include:

- hledger.org and the docs have been refreshed a few times, and now include many examples
- `hledger`'s code repo and bug tracker have moved from darcs/darcs hub/google code to git/github
- `hledger` has its own IRC channel on freenode: `#hledger`, with logging and commit/issue/travis notifications

`hledger` is available from hledger.org, github, hackage, stackage, and is packaged for a number of systems including Debian, Ubuntu, Gentoo, Fedora, and NixOS. See http://hledger.org/download or http://hledger.org/developer-guide for guidance.

Immediate plans:

- improve docs and help,
- improve parser speed and memory efficiency,
- integrate a separate parser for Ledger files built by John Wiegley,
- hledger-ui improvements,
- and work towards the 1.0 release.

Further reading

http://hledger.org

7.12.5 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!

The program works on Linux, Windows, and thanks to a contribution by Vincent Rasneur, it now also works on MacOS X.

Further reading

- http://arbtt.nomeata.de/
- http://www.joachim-breitner.de/blog/archives/336-The-Automatic-Rule-Based-Time-Tracker.html
- http://arbtt.nomeata.de/doc/users_guide/

7.12.6 Transient

Report by: Alberto Gómez Corona
Status: active development

Transient is a monad/applicative/Alternative with batteries included that bringing the power of high level effects in order to reduce the learning curve and make the haskell programmer productive. Effects include event handling/reactive, backtracking, extensible state, indetermism, concurrency, parallelism, thread control and distributed computing, publish/suscribe and client/server side web programming among others.

What is new in this report is:

- Distributed computing primitives have been moved to a different package: Transient-Universe.
- All distributed primitives derive from two basic ones: wormhole and teleport.
- Web browsers act as transient nodes participating in distributed computing. The transient program compiled with ghcjs is sent to the web browser when the URL point to a transient node with the HTTP protocol. Once the program is loaded the web node interact with the server nodes using webSocket communication using the same transient distributed primitives.
- Nodes in Web Browsers can render dynamic HTML widgets. The widget DSL is in the package ghcjs-hplay. It is a translation of the package hplayground.
- Events triggered int he web browser by the user can trigger server computations that send results back and are renderdered in the browser. So transient can create client/server widgets and these widgets can be combined with standard haskell combinators to create more complex ones.

Future work: server-side HTML rendering, making transient industry strenght, benchmarking, improve resource management, create web site.

Further reading

- Transient tutorial
- distributed Transient, GIT repository
- GIT repository of the widget rendering DSL
- Transient GIT repository
- An EDSL for Hard-working IT programmers
- The hardworking programmer II: practical backtracking to undo actions
- Publish-suscribe variables
- Moving processes between nodes
- Parallel non-determinism
- streamimg, distributed streaming, mapReduce with distributed datasets

7.12.7 tttool

Report by: Joachim Breitner
Status: active development

The Ravensburger Tiptoi® pen is an interactive toy for kids aged 4 to 10 that uses OiD technology to react when pointed at the objects on Ravensburger's Tiptoi books, games, puzzles and other toys. It is programmed via binary files in a proprietary, undocumented data format.

We have reverse engineered the format, and created a tool to analyze these files and generate our own. This program, called `tttool`, is implemented in Haskell, which turned out to be a good choice: Thanks to Haskell's platform independence, we can easily serve users on Linux, Windows and OS X.

The implementation makes use of some nice Haskell idoms such as a monad that, while parsing a binary, creates a hierarchical description of it and a writer monad that uses lazyness and MonadFix to reference positions in the file "before" these are determined.

Further reading

- https://github.com/entropia/tip-toi-reveng
- http://tttool.entropia.de/ (in German)
- http://funktionale-programmierung.de/2015/04/15/monaden-reverse-engineering.html (in German)

7.12.8 gipeda

Report by:	Joachim Breitner
Status:	active development

Gipeda is a a tool that presents data from your program's benchmark suite (or any other source), with nice tables and shiny graphs. Its name is an abbreviation for "Git performance dashboard" and highlights that it is aware of git, with its DAG of commits.

Gipeda powers the GHC performance dashboard at http://perf.haskell.org, but it builds on Shake and creates static files, so that hosting a gipeda site is easily possible. Also, it is useful not only for benchmarks: The author uses it to track the progress of his thesis, measured in area covered by the ink.

Further reading

https://github.com/nomeata/gipeda

7.12.9 Octohat (Stack Builders)

Report by:	Stack Builders
Participants:	Juan Carlos Paucar, Sebastian Estrella, Juan Pablo Santos
Status:	Working, well-tested minimal wrapper around GitHub's API

Octohat is a comprehensively test-covered Haskell library that wraps GitHub's API. While we have used it successfully in an open-source project to automate granting access control to servers, it is in very early development, and it only covers a small portion of GitHub's API.

Octohat is available on Hackage, and the source code can be found on GitHub.

We have already received some contributions from the community for Octohat, and we are looking forward to more contributions in the future.

Further reading

- https://github.com/stackbuilders/octohat
- Octohat announcement
- Octohat update

7.12.10 git-annex

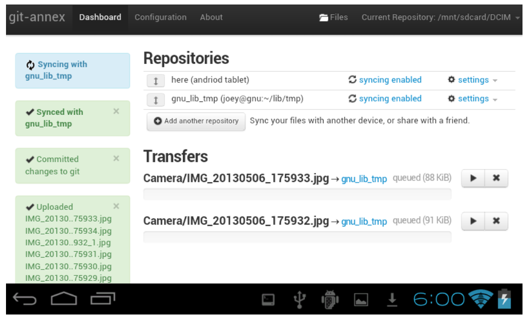
Report by:	Joey Hess
Status:	stable, actively developed

git-annex allows managing files with git, without checking the file contents into git. While that may seem paradoxical, it is useful when dealing with files larger than git can currently easily handle, whether due to limitations in memory, time, or disk space.

As well as integrating with the git command-line tools, git-annex includes a graphical app which can be used to keep a folder synchronized between computers. This is implemented as a local webapp using yesod and warp.

git-annex runs on Linux, OSX and other Unixes, and has been ported to Windows. There is also an incomplete but somewhat usable port to Android.

Five years into its development, git-annex has a wide user community. It is being used by organizations for purposes as varied as keeping remote Brazilian communities in touch and managing Neurological imaging data. It is available in a number of Linux distributions, in OSX Homebrew, and is one of the most downloaded utilities on Hackage. It was my first Haskell program.



At this point, my goals for git-annex are to continue to improve its foundations, while at the same time keeping up with the constant flood of suggestions from its user community, which range from adding support for storing files on more cloud storage platforms (around 20 are already supported), to improving its usability for new and non technically inclined users, to scaling better to support Big Data, to improving its support for creating metadata driven views of files in a git repository.

At some point I'd also like to split off any one of a half-dozen general-purpose Haskell libraries that have grown up inside the git-annex source tree.

Further reading

http://git-annex.branchable.com/

7.12.11 openssh-github-keys (Stack Builders)

Report by:	Stack Builders
Participants:	Justin Leitgeb
Status:	A library to automatically manage SSH access to servers using GitHub teams

It is common to control access to a Linux server by changing public keys listed in the `authorized_keys` file. Instead of modifying this file to grant and revoke access, a relatively new feature of OpenSSH allows the

accepted public keys to be pulled from standard output of a command.

This package acts as a bridge between the OpenSSH daemon and GitHub so that you can manage access to servers by simply changing a GitHub Team, instead of manually modifying the `authorized_keys` file. This package uses the Octohat wrapper library for the GitHub API which we recently released.

openssh-github-keys is still experimental, but we are using it on a couple of internal servers for testing purposes. It is available on Hackage and contributions and bug reports are welcome in the GitHub repository.

While we don't have immediate plans to put openssh-github-keys into heavier production use, we are interested in seeing if community members and system administrators find it useful for managing server access.

Further reading

https://github.com/stackbuilders/openssh-github-keys

7.12.12 propellor

Report by:	Joey Hess
Status:	actively developed

Propellor is a configuration management system for Linux that is configured using Haskell. It fills a similar role as Puppet, Chef, or Ansible, but using Haskell instead of the ad-hoc configuration language typical of such software. Propellor is somewhat inspired by the functional configuration management of NixOS.

A simple configuration of a web server in Propellor looks like this:

```
webServer :: Host
webServer = host "webserver.example.com"
  & ipv4 "93.184.216.34"
  & staticSiteDeployedTo "/var/www"
    `requires` Apt.serviceInstalledRunning "apache2"
    `onChange` Apache.reloaded
staticSiteDeployedTo :: FilePath → Property NoInfo
```

There have been many benefits to using Haskell for configuring and building Propellor, but the most striking are the many ways that the type system can be used to help ensure that Propellor deploys correct and consistent systems. Beyond typical static type benefits, GADTs and type families have proven useful. For details, see the blog.

An eventual goal is for Propellor to use type level programming to detect at compile time when a host has eg, multiple servers configured that would fight over the same port. Moving system administration toward using types to prove correctness properties of the system.

Another exciting possibility is using Propellor to not only configure existing Linux systems, but to manage their entire installation process. This has already been prototyped in a surprisingly small amount of added code (under 200 lines), which can replace arbitrary Linux systems with clean re-installs described entirely by Propellor's config.hs.

Further reading

http://propellor.branchable.com/

7.12.13 dimensional: Statically Checked Physical Dimensions

Report by:	Douglas McClean
Participants:	Björn Buckwalter, Alberto Valverde GonzÃąlez
Status:	active

Dimensional is a library providing data types for performing arithmetic with physical quantities and units. Information about the physical dimensions of the quantities/units is embedded in their types, and the validity of operations is verified by the type checker at compile time. The boxing and unboxing of numerical values as quantities is done by multiplication and division with units. The library is designed to, as far as is practical, enforce/encourage best practices of unit usage within the frame of the SI. Example:

```
d :: Fractional a ⇒ Time a → Length a
d t = a / _2 * t ^ pos2
  where a = 9.82 *~ (meter / second ^ pos2)
```

We are pleased to announce the release of dimensional 1.0, based on the prototype dimensional-dk implementation. Using data kinds and closed type families, the new version includes improved Haddock documentation, unit names with many options for pretty-printing, exact conversion factors between units (even between degrees and radians!), types for manipulating units and quantities whose dimensions are not known statically, and support for unboxed vectors.

New users with access to GHC 7.8 or later are strongly encouraged to use dimensional 1.0.

The "classic" dimensional library as released in 2006 is based on multi-parameter type classes and functional dependencies. It is stable with units being added on an as-needed basis. The primary documentation is the literate Haskell source code. Any future maintenance releases will have version numbers < 1.0.

The dimensional-tf library released in January 2012 a port of dimensional using type families will continue to be supported but is not recommended for new development.

Further reading

https://github.com/bjornbm/dimensional-dk

7.12.14 igrf: The International Geomagnetic Reference Field

Report by:	Douglas McClean
Status:	active

The igrf library provides a Haskell implementation of the International Geomagnetic Reference Field, including the latest released model values.

Upcoming development efforts include a parser for the model files as released by the IAGA and a dimensionally-typed interface using the dimensional library.

Further reading

https://github.com/dmcclean/igrf

7.12.15 Haskell in Green Land

Report by:	Gilberto Melfe
Participants:	Luís Gabriel Lima, Francisco Soares-Neto, Paulo Lieuthier, Fernando Castor, João Paulo Fernandes
Status:	mostly stable, with ongoing extensions

In the Haskell in Green Land initiative we attempt to understand the energy behavior of programs written in Haskell. It is particularly interesting to study Haskell in the context of energy consumption since Haskell has mature implementations of sophisticated features such as laziness, partial function application, software transactional memory, tail recursion, and a kind system. Furthermore, recursion is the norm in Haskell programs and side effects are restricted by the type system of the language.

We analyze the energy efficiency of Haskell programs from two different perspectives:

a) strictness: by default, expressions in Haskell are lazily evaluated, meaning that any given expression will only be evaluated when it is first necessary. This is different from most programming languages, where expressions are evaluated strictly and possibly multiple times;

b) concurrency: previous work has demonstrated that concurrent programming constructs can influence energy consumption in unforeseen ways.

Concretely, we have addressed the following high-level research question: To what extent can we save energy by refactoring existing Haskell programs to use different data structure implementations or concurrent programming constructs?

In order to address this research question, we conducted two complementary empirical studies:

a) we analyzed the performance and energy behavior of several benchmark operations over 15 different implementations of three different types of data structures considered by the Edison Haskell library;

b) we assessed three different thread management constructs and three primitives for data sharing using nine benchmarks and multiple experimental configurations.

Overall, experimental space exploration comprises more than 2000 configurations and 20000 executions.

We found that small changes can make a big difference in terms of energy consumption. For example, in one of our benchmarks, under a specific configuration, choosing one data sharing primitive over another can yield 60% energy savings. Nonetheless, there is no universal winner.

In addition, the relationship between energy consumption and performance is not always clear. Generally, especially in the sequential benchmarks, high performance is a proxy for low energy consumption. Nonetheless, when concurrency comes into play, we found scenarios where the configuration with the best performance (30% faster than the one with the worst performance) also exhibited the second worst energy consumption (used 133% more energy than the one with the lowest usage).

To support developers in better understanding this complex relationship, we have extended two existing tools from the Haskell world:

i) the powerful benchmarking library Criterion;

ii) the profiler that comes with the Glasgow Haskell Compiler.

Originally, such tools were devised for performance analysis and we have adapted them to make them *energy-aware*.

Further reading

The data for this study, the source code for the implemented tools and benchmarks as well as a paper describing all the details of our work can be found at green-haskell.github.io.
Furthermore, we have referenced the following papers:

- Pinto, Gustavo and Castor, Fernando and Liu, Yu David: *Understanding Energy Behaviors of Thread Management Constructs*, Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications
- Luís Gabriel Lima and Gilberto Melfe and Francisco Soares-Neto and Paulo Lieuthier and João Paulo Fernandes and Fernando Castor, *Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language*, Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'2016)

7.12.16 Kitchen Snitch server

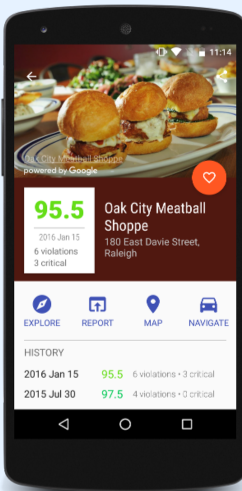
Report by:	Dino Morelli
Participants:	Betty Diegel
Status:	stable, actively developed

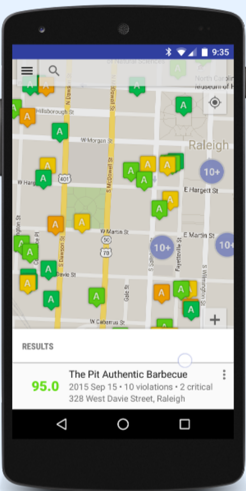
This project is the server-side software for Kitchen Snitch, a mobile application that provides health inspection scores, currently for the Raleigh-Durham area in NC, USA. The data can be accessed on maps along with inspection details, directions and more.

The back-end software provides a REST API for mobile clients and runs services to perform regular inspection data acquisition and maintenance.

Kitchen Snitch has been in development for over a year and is running on AWS. The mobile client and server were released for public use in April of 2016 after a beta-test period.

Some screenshots of the Android client software in action:





Getting Kitchen Snitch:

The mobile client can be installed from the Google Play Store. There is also a landing page http://getks.honuapps.com/.

The Haskell server source code is available on darcshub at the URLs below.

Further reading

- ks-rest http://hub.darcs.net/dino/ks-rest
- ks-download http://hub.darcs.net/dino/ks-download
- ks-library http://hub.darcs.net/dino/ks-library

7.12.17 DSLsofMath

Report by:	Patrik Jansson
Participants:	Cezar Ionescu, Irene Lobo Valbuena, Adam Sandberg Ericsson
Status:	active development

"Domain Specific Languages of Mathematics" is a project at Chalmers University of Technology developing a new BSc level course and accompanying material for learning and applying classical mathematics (mainly real and complex analysis). The main idea is to encourage the students to approach mathematical domains from a functional programming perspective: to identify the main functions and types involved and, when necessary, to introduce new abstractions; to give calculational proofs; to pay attention to the syntax of the mathematical expressions; and, finally, to organize the resulting functions and types in domain-specific languages.

The first instance of the course was carried out Jan-March 2016 at Chalmers and the course material is available on github. The next step is to write up the lecture notes as a book during the autumn, in preparation for the next instance of the course early 2017. Contributions and ideas are welcome!

Further reading

- DSLsofMath (github organisation)
- TFPIE 2015 paper
- Exam 2016 with solutions

8 Commercial Users

8.1 Well-Typed LLP

Report by:	Andres Löh
Participants:	Duncan Coutts, Adam Gundry

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⟩.

We have been working on a large number of different projects for various clients, most of which are unfortunately not publically visible.

Here is a non-exhaustive list of open-source contributions we have made:

Ben Gamari and Austin Seipp have been helping with the GHC 8.0 release (→ 3.1).

Duncan Coutts and Austin Seipp have put a lot of effort into the `binary-serialise-cbor` library, which is intended to be an improved version of (large parts of) the `binary` package. With the help of many other contributors, the package is now nearly release-ready.

Duncan Coutts has been working on bringing nix-style local builds to Cabal (→ 6.3.1).

Andres Löh has helped with the development of the Haskell Servant web framework, continuing to improve routing. This work is included in the 0.7 release.

In October 2016, we are planning to organize Haskell courses in London as well as the Haskell eXchange again. There will be a public call for contributions soon.

If you are interested in getting information about Well-Typed events (such as conferences or courses we are participating in or organizing), you can subscribe to a new mailing list at http://www.well-typed.com/cgi-bin/mailman/listinfo/events.

We are also always looking for new clients and projects, so if you have something we could help you with, or even would just like to tell us about your use of Haskell, please just drop us an e-mail.

Further reading

- Company page: http://www.well-typed.com
- Blog: http://blog.well-typed.com/
- `binary-serialise-cbor` package: https://github.com/well-typed/binary-serialise-cbor/
- Austin talk on `binary-serialise-cbor` at Boston Haskell Meetup: https://www.youtube.com/watch?v=Mj2cXQXgyWE
- Servant: http://haskell-servant.readthedocs.io/
- Haskell eXchange 2016 (registration): https://skillsmatter.com/conferences/7276-haskell-exchange-2016
- Haskell courses in London: https://skillsmatter.com/explore?content=courses&location=&q=tag%3Ahaskell
- Training page: http://www.well-typed.com/services_training
- Well-Typed events mailing list: http://www.well-typed.com/cgi-bin/mailman/listinfo/events

8.2 Bluespec Tools for Design of Complex Chips and Hardware Accelerators

Report by:	Rishiyur Nikhil
Status:	Commercial product; free for academia

Bluespec, Inc. provides an industrial-strength language (BSV) and tools for high-level hardware design. Components designed with these are shipping in some commercial smartphones and tablets today.

BSV is used for all aspects of ASIC and FPGA design — specification, synthesis, modeling, and verification. Digital circuits are *described* using a notation with Haskell semantics, including algebraic types, polymorphism, type classes, higher-order functions and monadic elaboration. Strong static checking is also used to support discipline for multiple clock-domains and gated clocks. The dynamic semantics of a such circuits are described using Term Rewriting Systems (which are essentially atomic state transitions). BSV is applicable to all kinds of hardware systems, from algorithmic "datapath" blocks to complex control blocks such as processors, DMAs, interconnects, and caches, and to complete SoCs (Systems on a Chip).

Perhaps uniquely among hardware-design languages, BSV's rewrite rules enable design-by-refinement, where an initial executable approximate design is systematically transformed into a quality implementation by successively adding functionality and architectural detail.

Before synthesizing to hardare, a circuit description can be executed and debugged in *Bluesim*, a fast simulation tool. Then, the *bsc* tool compiles BSV into high-quality Verilog, which is then further synthesized into netlists for ASICs and FPGAs using standard synthesis tools. There are extensive libraries and infrastructure components to make it easy to build FPGA-based accelerators for compute-intensive software.

Bluespec also provides implementations and development environments for CPUs based on the U.C. Berkeley RISC-V instruction set (www.riscv.org).

Status and availability

BSV tools have been available since 2004, both commercially and free for academic teaching and research. It is used in a several leading universities (incl. MIT, U.Cambridge, and IIT Chennai) for computer architecture research.

Further reading

- *Types, Functional Programming and Atomic Transactions in Hardware Design*, R.S. Nikhil, in *In Search of Elegance in the Theory and Practice of Computation, Essays dedicated to Peter Buneman (Festschrift), Springer-Verlag Lecture Notes in Computer Science, LNCS 8000*, pp.418-431, 2013.
- *Abstraction in Hardware System Design*, R.S. Nikhil, in *Communications of the ACM*, 54:10, October 2011, pp. 36-44.
- *BSV by Example*, R.S. Nikhil and K. Czeck, 2010, book available on Amazon.com (or free PDF from Bluespec, Inc.)
- http://bluespec.com/SmallExamples/index.html: from *BSV by Example*.
- http://www.cl.cam.ac.uk/~swm11/examples/bluespec/: Simon Moore's BSV examples (U. Cambridge).
- http://csg.csail.mit.edu/6.375: *Complex Digital Systems*, MIT courseware.

8.3 Better

Report by:	Carl Baatz

Better provides a platform for delivering adaptive online training to students and employees.

Companies and universities work with us to develop courses which are capable of adapting to individual learners. This adaptivity is based on evidence we collect about the learner's understanding of the course material (primarily by means of frequent light-weight assessments). These courses run on our platform, which exposes a (mobile-compatible) web interface to learners. The platform also generates course statistics so that managers/teachers can monitor the progress of the class taking the course and evaluate its effectiveness.

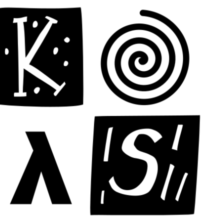
The backend is entirely written in Haskell. We use the `snap` web framework and we have a storage layer written on top of `postgres-simple` which abstracts data retrieval, modification, and versioning. The choice of language has worked out well for us: as well as the joy of writing Haskell for a living, we get straightforward deployment and extensive server monitoring courtesy of `ekg`. Using GHC's profiling capabilities, we have also managed to squeeze some impressive performance out of our deployment.

The application-specific logic is all written in Haskell, as is *most* of the view layer. As much rendering as possible is performed on the backend using `blaze-html`, and the results are sent to a fairly thin single-page web application written in Typescript (which, while not perfect, brings some invaluable static analysis to our front-end codebase).

The company is based in Zurich, and the majority of the engineering team are Haskellers. We enjoy a high level of involvement with the Zurich Haskell community and are delighted to be able to host the monthly HaskellerZ user group meetups and the yearly ZuriHac hackathon.

8.4 Keera Studios LTD

Report by:	Ivan Perez



Keera Studios Ltd. is a game development studio that uses Haskell to create mobile and desktop games. We have published Magic Cookies!, the first commercial game for Android written in Haskell, now available on Google Play™ (https://goo.gl/cM1tD8).

We have also shown a breakout-like game running on a Android tablet (http://goo.gl/53pK2x), using *hardware acceleration* and *parallelism*. The desktop version of this game additionally supports Nintendo Wiimotes and Kinect. This proves that Haskell truly is viable option for *professional game development*, both for mobile and for desktop. A new game is currently being developed for Android and iOS.

In order to provide more reliable code for our clients, we have developed a battery of small Haskell mobile apps, each testing only one feature. We have dozens of apps, covering SDL and multimedia including multitouch support, accelerometers, and stereoscopy (for more realistic depth and 3D effects). Our battery also includes apps that communicate with Java via C/C++,

used for Facebook/Twitter status sharing, to save game preferences using Android's built-in Shared Preferences storage system, or to create Android widgets. We have also started the Haskell Game Programming project http://git.io/vlxtJ, which contains documentation and multiple examples of multimedia, access to gaming hardware, physics and game concepts. We continue to participate in Haskell meetings and engaging in the community, with a recent talk on Game Programming at the Haskell eXchange 2015.

We have developed GALE, a DSL for graphic adventures, together with an engine and a basic IDE that allows non-programmers to create their own 2D graphic adventure games without any knowledge of programming. Supported features include multiple character states and animations, multiple scenes and layers, movement bitmasks (used for shortest-path calculation), luggage, conversations, sound effects, background music, and a customizable UI. The IDE takes care of asset management, generating a fully portable game with all the necessary files. The engine is multiplatform, working *seamlessly* on Linux, Windows and Android. We are continue beta-testing GALE games on Google Play.

We have released Keera Hails, the *reactive* library we use for desktop GUI applications, as Open Source (http://git.io/vTvXg). Keera Hails is being *actively developed* and provides integration with Gtk+, network sockets, files, FRP Yampa signal functions and other external resources. Experimental integration with wxWidgets and Qt is also available, and newer versions include partial backends for Android (using Android's default widget system, communicating via FFI) and HTML DOM (via GHCJS). We are working on providing complete backends for all major GUI toolkits and platforms. Recent updates to our project are geared towards adding documentation, tests and benchmarks, in order to facilitate using, understanding and extending the framework and guaranteeing a high level of quality.

Apart from implementing a simple yet powerful form of reactivity, Keera Hails addresses common problems in Model-View-Controller, providing an application skeleton with a scalable architecture and thread-safe access to the application's internal model. Accompanying libraries feature standarised solutions for common features such as configuration files and internationalisation. We have used this framework in commercial applications (including but not limited to GALE IDE), and in the Open-Source posture monitor Keera Posture (http://git.io/vTvXy). Links to these applications, examples, demos and papers, including a recent paper on Reactive Values and Relations presented at the Haskell Symposium 2015, are available on our website.

We are committed to using Haskell for all our operations. For games we often opt for the Arrowized Functional Reactive Programming Domain-Specific Language Yampa (http://git.io/vTvxQ) or for Keera GALE. For desktop GUI applications we use our own Keera Hails (http://git.io/vTvXg). To create web applications and internal support tools we use Yesod, and continue developing our project management, issue tracking and invoicing web application to facilitate communication with our clients.

Screenshots, videos and details are published regularly on our Facebook page (https://www.facebook.com/keerastudios) and on our company website (http://www.keera.co.uk). If you want to use Haskell in your next game, desktop or web application, or to receive more information, please contact us at .

8.5 Stack Builders

Report by:	Stack Builders
Status:	software consultancy

Stack Builders is an international Haskell and Ruby agile software consultancy with offices in New York, United States, and Quito, Ecuador.

In addition to our Haskell software consultancy services, we are actively involved with the Haskell community:

- We organize Quito Lambda, a monthly meetup about functional programming in Quito, Ecuador.
- We maintain several packages in Hackage including hapistrano, inflections, octohat, openssh-github-keys, and twitter-feed.
- We talk about Haskell at universities and events such as Lambda Days and BarCamp Rochester.
- We write blog posts and tutorials about Haskell.

For more information, take a look at our website or get in touch with us at info@stackbuilders.com.

Further reading

http://www.stackbuilders.com/

8.6 Optimal Computational Algorithms, Inc.

Report by:	Christopher Anand

OCA develops high-performance, high-assurance mathematical software using Coconut (COde CONstructing User Tool), a hierarchy of DSLs embedded in Haskell, which were originally developed at McMaster University. The DSLs encode declarative assembly language, symbolic linear algebra, and algebraic transformations. Accompanying tools include interpreters, simulators, instruction schedulers, code transformers (both rule-based and ad-hoc) and graph and schedule visualizers.

To date, Coconut math function libraries have been developed for five commercial architectures. Taking advantage of Cocont's symbolic code generation, software for reconstructing multi-coil Magnetic Resonance Images was generated from a high-level mathematical specification. The implementation makes full use of dual-CPUs, multiple cores and SIMD parallelism, and is licensed to a multi-national company. The specification is transformed using rules for symbolic differentiation, algebraic simplification and parallelization. The soundness of the generated parallelization can be verified in linear time (measured with respect to program size).

Further reading

- http://www.cas.mcmaster.ca/~kahl/Publications/TR/Anand-Kahl-2007a_DSL/
- http://www.cas.mcmaster.ca/~anand/papers/AnandKahlThaller2006.pdf
- http://www.cas.mcmaster.ca/sqrl/papers/SQRLreport50.pdf
- https://macsphere.mcmaster.ca/handle/11375/10755
- http://www.cas.mcmaster.ca/~anand/papers/CAS-14-05-CA.pdf

8.7 Snowdrift.coop

Report by:	Bryan Richter
Participants:	Aaron Wolf et al.
Status:	Work in progress

Snowdrift.coop is a web platform for funding and supporting free/libre/open projects. We are tackling the 'snowdrift dilemma' that limits contributions to non-rivalrous goods such as open-source software. The organization is a non-profit multi-stakeholder cooperative, and all code is available under OSI- and FSF-approved licenses. Haskell is our primary programming language, and we welcome any interested contributors to help us accelerate our progress.

In our current work we have recently focused on three main areas: 1) opening the project to greater participation through code refactoring and tool development, 2) firming up the co-op governance structure, and 3) creating a comprehensive design framework for the website. There is also plenty of ongoing feature development on various aspects of the live site.

One notable contribution Snowdrift has made to the Haskell ecosystem is a thorough 'getting started' experience for beginners, from text editor suggestions to introductions to git. As part of that effort, we have developed a foolproof build process, with a tip of our hats to the new tool `stack`, and have developed a database initialization tool and various Yesod integrations with ghci and text editors. Interested contributors will find many opportunities for progress in this area.

The funding mechanism is not yet functional but progressing. Once functional, Snowdrift.coop itself will be a supported project, and should prove to be an excellent test-case for the adoption and success of the concept. In the meanwhile, we are actively looking for ways to improve both productivity and opportunities for our distributed team of volunteers. Experienced Haskellers are invited to mentor volunteers, take ownership of component libraries, and provide opinions and insights. New Haskellers—not to mention designers, writers, economists, legal professionals, or anyone else philosophically inclined to our mission of freeing the commons—are especially welcome; we pride ourselves on being inclusive and approachable by (non-)programmers at any level of technical sophistication!

Further reading

- https://snowdrift.coop
- https://lists.snowdrift.coop
- https://git.gnu.io/snowdrift

8.8 McMaster Computing and Software Outreach

Report by:	Christopher Anand
Status:	active

McMaster Computing and Software Outreach visits schools in Ontario, Canada to teach basic Computer Science topics and discuss the impacts of the Information Revolution, teaching children from six to sixteen. In 2015, we swapped out Python in our programming activities for ELM, which is a functional replacement for JavaScript. ELM looks a lot like Haskell, but does not have user-definable type classes and is strict. Thanks largely to ELM, we tripled the number of children in our workshops to 3500. Our hypothesis is that declarative programming matches the computational model instructed in basic algebra, which receives significant attention in the Ontario curriculum. But it is possible that other aspects of the language are more important. We also believe that immediate graphical feedback is important, as do many other educators, but since declarative specifications of vector graphics are significantly simpler than stateful constructions, these issues are not orthogonal.

We would like to thank Evan Czapliki for creating ELM, and assisting us.

To see what children with no programming experience can accomplish in a declarative language in a just a few hours, please visit http://outreach.mcmaster.ca/menu/fame.html. Note that grade 4 students are about ten years old.

9 Research and User Groups

9.1 Haskell at Eötvös Loránd University (ELTE), Budapest

Report by:	PÁLI Gábor János
Status:	ongoing

Education

There are many different courses on functional programming – mostly taught in Haskell – at Eötvös Loránd University, Faculty of Informatics. Currently, we are offering the following courses in that regard:

- Functional programming for first-year Hungarian undergraduates in Software Technology and second-year Hungarian teacher of informatics students, both as part of their official curriculum.
- An additional semester on functional programming with Haskell for bachelor's students, where many of the advanced concepts are featured, such as algebraic data types, type classes, functors, monads and their use. This is an optional course for Hungarian undergraduate and master's students, supported by the Eötvös József Collegium.
- Functional programming for Hungarian and foreign-language master's students in Software Technology. The curriculum assumes no prior knowledge on the subject in the beginning, then through teaching the basics, it gradually advances to discussion of parallel and concurrent programming, property-based testing, purely functional data structures, efficient I/O implementations, embedded domain-specific languages, and reactive programming. It is taught in both one- and two-semester formats, where the latter employs the Clean language for the first semester.

In addition to these, there is also a Haskell-related course, Type Systems of Programming Languages, taught for Hungarian master's students in Software Technology. This course gives a more formal introduction to the basics and mechanics of type systems applied in many statically-typed functional languages.

For teaching some of the courses mentioned above, we have been using an interactive online evaluation and testing system, called ActiveHs. It contains several dozens of systematized exercises, and through that, some of our course materials are available there in English as well.

Our homebrew online assignment management system, "BE-AD" keeps working on for the fourth semester starting from this September. The BE-AD system is implemented almost entirely in Haskell, based on the Snap web framework and Bootstrap. Its goal to help the lecturers with scheduling course assignments and tests, and it can automatically check the submitted solutions as an option. It currently has over 700 users and it provides support for 12 courses at the department, including all that are related to functional programming. This is still in an alpha status yet so it is not available on Hackage as of yet, only on GitHub, but so far it has been performing well, especially in combination with ActiveHs.

Further reading

- Haskell course materials (in English): http://pnyf.inf.elte.hu/fp/Index_en.xml
- Agda tutorial (in English): http://people.inf.elte.hu/pgj/agda/tutorial/
- ActiveHs: http://hackage.haskell.org/package/activehs
- BE-AD: http://github.com/andorp/bead

9.2 Artificial Intelligence and Software Technology at Goethe-University Frankfurt

Report by:	David Sabel
Participants:	Manfred Schmidt-Schauß

Semantics of Functional Programming Languages. Extended call-by-need lambda calculi model the semantics of Haskell. We analyze the semantics of those calculi with a special focus on the correctness of program analyses and program transformations. In our recent research, we use Haskell to develop automated tools to show correctness of program transformations, where the method is syntax-oriented and computes so-called forking and commuting diagrams by a combination of several unification algorithms.

Improvements In recent research we analyzed whether program transformations are optimizations, i.e. whether they improve the time resource behavior. We showed that common subexpression elimination is an improvement, also under polymorphic typing. We developed methods for better reasoning about improvements in the presence of sharing, i.e. in call-by-need calculi. Ongoing work is to enhance the techniques to (preferably automatically) verify that program transformations are improvements.

Concurrency. We analyzed a higher-order functional language with concurrent threads, monadic IO, MVars and concurrent futures which models Concurrent Haskell. We proved that this language conservatively extends the purely functional core of Haskell. In a similar program calculus we proved correctness of a

highly concurrent implementation of Software Transactional Memory (STM) and developed an alternative implementation of STM Haskell which performs quite early conflict detection.

Grammar based compression. This research topic focuses on algorithms on grammar compressed data like strings, matrices, and terms. Our goal is to reconstruct known algorithms on uncompressed data for their use on grammars without prior decompression. We implemented several algorithms as a Haskell library including efficient algorithms for fully compressed pattern matching.

Cycle Rewriting. Cycle rewrite applies string rewriting to cycles – a cycle is a string where start and end are connected. We developed techniques to prove cycle termination. A tool (cycsrs) was implemented in Haskell to show termination by transformation and then using automated termination provers. Also tools to prove cycle termination by trace-decreasing matrix interpretations and to disprove termination by a smart search for counter-examples were implemented in Haskell. It participated in the Termination Competition 2015.

Further reading

http://www.ki.informatik.uni-frankfurt.de/research/HCAR.html

9.3 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, we use for example also Erlang and ML, Haskell provides a major focus and common language for teaching and research.

Our members pursue a variety of Haskell-related projects, several of which are reported in other sections of this report. Stephen Adams is working on advanced refactoring of Haskell programs, extending HaRe. Andreas Reuleaux is building in Haskell a refactoring tool for a dependently typed functional language. Maarten Faddegon is working on making tracing for Haskell practical and easy to use by building the lightweight tracer and debugger Hoed. Olaf Chitil is also working on tracing, including the further development of the Haskell tracer Hat, and on type error debugging. Meng Wang is working on lenses, bidirectional transformation and property-based testing (QuickCheck). Scott Owens is working on verified compilers for the (strict) functional language CakeML. He and Simon Thompson are also working on verified refactoring. Recently Stefan Kahrs worked on minimising regular expressions, implemented in Haskell.

We are always looking for more PhD students. We are particularly keen to recruit students interested in programming tools for verification, tracing, refactoring, type checking and any useful feedback for a programmer. The school and university have support for strong candidates: more details at http://www.cs.kent.ac.uk/pg or contact any of us individually by email.

We are also keen to attract researchers to Kent to work with us. There are many opportunities for research funding that could be taken up at Kent, as shown in the website http://www.kent.ac.uk/researchservices/sciences/fellowships/index.html. Please let us know if you're interested in applying for one of these, and we'll be happy to work with you on this.

Finally, if you would like to visit Kent, either to give a seminar if you're passing through London or the UK, or to stay for a longer period, please let us know.

Further reading

- PLAS group:
  http://www.cs.kent.ac.uk/research/groups/plas/
- Maarten Faddegon and Olaf Chitil: Lightweight Computation Tree Tracing for Lazy Functional Languages. PLDI 2016.
- Haskell: the craft of functional programming:
  http://www.haskellcraft.com
- Refactoring Functional Programs: http://www.cs.kent.ac.uk/research/groups/plas/hare.html
- Hoed, a lightweight Haskell tracer and debugger:
  https://github.com/MaartenFaddegon/Hoed
- Hat, the Haskell Tracer:
  http://projects.haskell.org/hat/
- CakeML, a verification friendly dialect of SML:
  https://cakeml.org
- Heat, an IDE for learning Haskell:
  http://www.cs.kent.ac.uk/projects/heat/

9.4 Haskell at KU Leuven, Belgium

Report by: Tom Schrijvers

Functional Programming, and Haskell in particular, is an active topic of research and teaching in the Declarative Languages & Systems group of KU Leuven, Belgium.

Teaching Haskell is an integral part of the curriculum for both informatics bachelors and masters of engineering in computer science. In addition, we offer and supervise a range of Haskell-related master thesis topics.

Research We actively pursue various Haskell-related lines of research. Some recent and ongoing work:

- Steven Keuchel works on InBound, a Haskell-like DSL for specifying abstract syntax trees with binders.
- George Karachlias works on extending GHC's pattern match checker to deal with GADTs, in collaboration with Dimitrios Vytiniotis and Simon Peyton Jones.
- Alexander Vandenbroucke extends the nondeterminism monad with tabulation, a form of memoization "on steroids" from logic programming.
- With Nicolas Wu we have recently worked on fusion for free monads to obtain efficient algebraic effect handlers. See our forthcoming MPC 2015 paper.
- With Mauro Jaskelioff and Exequiel Rivas we launch a new slogan:

  > Nondeterminism monads are just near-semirings in the category of endofunctors, what's the problem?

  See our forthcoming paper at PPDP 2015.

Leuven Haskell User Group We host the Leuven Haskell User Group, which has held its first meeting on March 3, 2015. The group meets roughly every other week and combines formal presentations with informal discussion. For more information: http://groups.google.com/forum/#!forum/leuven-haskell

Further reading

http://people.cs.kuleuven.be/~tom.schrijvers/Research/

9.5 HaskellMN

Report by:	Kyle Marek-Spartz
Participants:	Tyler Holien
Status:	ongoing

HaskellMN is a user group from Minnesota. We have monthly meetings on the third Wednesday in downtown Saint Paul.

Further reading

http://www.haskell.mn

9.6 Functional Programming at KU

Report by:	Andrew Gill
Status:	ongoing

Functional Programming continues 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 Andrew Gill) together form the core functional programming initiative at KU. All the Haskell related KU projects are now focused on use-cases for the remote monad design pattern (→ 6.4.11). One example is the Haskino Project (→ 6.1.6).

Further reading

The Functional Programming Group: http://www.ittc.ku.edu/csdl/fpg

9.7 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. Members of the group include users of Haskell, Ocaml, LISP, Scala, F#, Scheme and others. We have 10 meetings per year (Feb–Nov) and meet on the fourth Wednesday of each month. We regularly get 40–50 attendees, with a 70/30 industry/research split. Talks this year have included material on compilers, theorem proving, type systems, Haskell web programming, dynamic programming, Scala and more. We usually have about 90 mins of talks, starting at 6:30pm. All welcome.

Further reading

- http://groups.google.com/group/fp-syd
- http://fp-syd.ouroborus.net/
- http://fp-syd.ouroborus.net/wiki/Past/2016

9.8 Regensburg Haskell Meetup

Report by: Andres Löh

Since autumn 2014 Haskellers in Regensburg, Bavaria, Germany have been meeting roughly once per month to socialize and discuss Haskell-related topics.

I'm happy to say that this meetup continues to thrive. We typically have between 10 and 15 attendees (which is really not bad if you consider the size of Regensburg), and we often get visitors from Munich, Nürnberg and Passau.

New members are always welcome, whether they are Haskell beginners or experts. If you are living in the area or visiting, please join! Meetings are announced a few weeks in advance on our meetup page: http://www.meetup.com/Regensburg-Haskell-Meetup/.

9.9 Curry Club Augsburg

Report by: Ingo Blechschmidt
Status: active

Since March 2015 haskellistas, scalafists, lambdroids, and other fans of functional programming languages in Augsburg, Bavaria, Germany have been meeting every four weeks in the OpenLab, Augsburg's hacker space. Usually there are ten to twenty attendees.

At each meeting, there are typically two to three talks on a wide range of topics of interest to Haskell programmers, such as latest news from the Kmettiverse and introductions to the category-theoretic background of freer monads. Afterwards we have stimulating discussions while dining together.

From time to time we offer free workshops to introduce new programmers to the joy of Haskell.

Newcomers are always welcome! Recordings of our talks are available at http://www.curry-club-augsburg.de/.

Further reading

http://www.curry-club-augsburg.de/

9.10 Italian Haskell Group

Report by: Francesco Ariis
Status: ongoing

Born in Summer 2015, the Italian Haskell Group is an effort to advocate functional programming and share our passion for Haskell through real-life meetings, discussion groups and community projects.

There have been 3 meetups (in Milan, Bologna and Florence), our plans to continue with a quarterly schedule. Anyone from the experienced hacker to the functionally curious newbie is welcome; during the rest of the year you can join us on our irc/mumble channel for haskell-related discussions and activities.

Further reading

- site: http://haskell-ita.it/
- IRC channel:
  https://webchat.freenode.net/?channels=%23haskell.it
- Discussion forum :
  https://groups.google.com/forum/#!forum/haskell_ita