Haskell Communities and Activities Report

http://tinyurl.com/haskcar

Thirty Fourth Edition — May 2018

Mihai Maruseac (ed.)

Chris Allen
Christopher Anand
Moritz Angermann
Francesco Ariis
Heinrich Apfelmus
Gershom Bazerman
Doug Beardsley
Jost Berthold
Ingo Blechschmidt
Sasa Bogicevic
Emanuel Borsboom
Jan Bracker
Jeroen Bransen
Joachim Breitner
Rudy Braquehais
Björn Buckwalter
Erik de Castro Lopo
Manuel M. T. Chakravarty
Eitan Chatav
Olaf Chitil
Alberto Gómez Corona
Nils Dallmeyer
Tobias Dammers
Kei Davis
Dimitri DeFigueiredo
Richard Eisenberg
Maarten Faddegon
Dennis Felsing
Olle Fredriksson
Phil Freeman
Marc Fontaine
PÁLI Gábor János
Michał J. Gajda
Ben Gamari
Michael Georgoulopoulos
Andrew Gill
Mikhail Glushenkov
Mark Grebe
Gabor Greif
Adam Gundry
Jennifer Hackett
Jurriaan Hage
Martin Handley
Bastiaan Heeren
Sylvain Henry
Joey Hess
Kei Hibino
Guillaume Hoffmann
Graham Hutton
Nicu Ionita
Judah Jacobson
Patrik Jansson
Wanqiang Jiang
Dzianis Kabanau
Nikos Karagiannidis
Anton Kholomiov
Oleg Kiselyov
Ivan Krišto
Yasuaki Kudo
Harendra Kumar
Rob Leslie
David Lettier
Ben Lippmeier
Andres Löh
Rita Loogen
Tim Matthews
Simon Michael
Andrey Mokhov
Dino Morelli
Damian Nadales
Henrik Nilsson
Wisnu Adi Nurcahyo
Ulf Norell
Ivan Perez
Jens Petersen
Sibi Prabakaran
Bryan Richter
Herbert Valerio Riedel
Alexey Radkov
Vaibhav Sagar
Kareem Salah
Michael Schröder
Christian Höner zu Siederdissen
Ben Sima
Jeremy Singer
Gideon Sireling
Erik Sjöström
Chris Smith
Michael Snoyman
David Sorokin
Lennart Spitzner
Yuriy Syrovetskiy
Jonathan Thaler
Henk-Jan van Tuyl
Tillmann Vogt
Michael Walker
Li-yao Xia
Kazu Yamamoto
Yuji Yamamoto
Brent Yorgey
Christina Zeller
Marco Zocca

Preface

This is the 34th edition of the Haskell Communities and Activities Report. This report has 148 entries, 5 more than in the previous edition. Of these, 39 projects have received substantial updates and 18 entries are completely new. 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.

Since my goal is to keep only entries which are under active development (defined as receiving an update in a 3-editions sliding window), contributions from May 2015 and before have been completely removed, excepting those related to user groups and communities. However, they can be resurfaced in the next edition, should a new update be sent for them. For the 30th edition, for example, we had around 20 new entries which resurfaced.

A call for new HCAR entries and updates to existing ones will be issued on the Haskell mailing lists in late September/early October.

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

Mihai Maruseac, LeapYear Technologies Inc., US
⟨hcar@haskell.org⟩

PS: For this edition, I would like to thank Robin Bate Boerop for mentioning HCAR in his talk at BayHac. A few of the new entries are the result of his announcement.

Contents

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 skill-set. 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 a manifold addition centered on Type Equality and Overlapping Instances.

Type Equality Predicates and Assertions

We describe several forms of testing two types for equality, and asserting that two types are equal or not equal. The predicates differ in how they compare type variables, whether they allow wildcards, etc.

Described predicates:

- TypeEq *t1 t2*: simplifies to HTrue if the type checker regards *t1* and *t2* as the same. The result is HFalse if the type checker can see the two types as different. The decision is postponed if *t1* and *t2* are not instantiated enough;
- TypeEqTotal *t1 t2*: immediately yields either HFalse or HTrue, regardless of how *t1* and *t2* are instantiated. A type variable is TypeEqTotal only to itself;
- *t1*∼*t2*: asserts that *t1* and *t2* are the same, and, as a *side effect*, instantiates type variables in *t1* and *t2* to make the type the same. If the types cannot be made the same, a type error is raised;
- A predicate version of *t1*∼*t2* – that is, TypeEq with the side-effect of instantiation of type variables – is incoherent;

Read the tutorial online.
How to implement type equality.

For and Against Overlapping Instances

Overlapping instances are so practically appealing because they express the common pattern of adding a special case to an existing set of overloaded functions. We illustrate the pattern on a real-life example of optimizing a generic library. The example demonstrates the conflict between two practically useful features, overlapping instances and associated data types.

Overlapping instances are controversial because they straddle a contradiction. They embody "negation as failure": the general type class instance is chosen for the given type only when all more specific instances failed to match the type. Negation-as-failure presupposes closed world, or a fixed set of instances. However, type classes are *open*: the user may add more instances at any time, in the same or different modules.

Read the tutorial online.
Another use case: Comparing types by their shape.
Objections to overlapping instances.

Type-level type introspection, equality and matching

We describe type-level representation of types: essentially type-level Typeable. The library lets us check if two types are equal or dis-equal, and compare them by shape. We may use wildcards in type comparisons.

The library exhibits type-level conditional and higher-order type families such as Member (which takes a type-level equality predicate as an argument).

Read the tutorial online.
Using TTypeable to avoid OverlappingInstances.

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.

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/

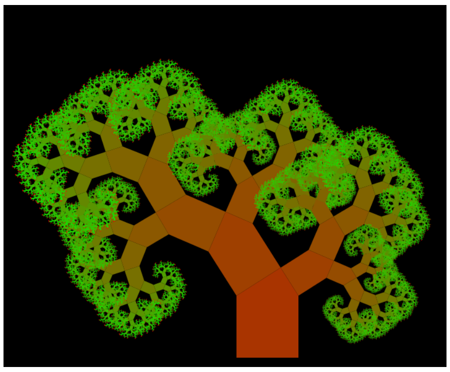
2.3 Learning Haskell

Report by:	Manuel M. T. Chakravarty
Participants:	Gabriele Keller
Status:	Work in progress with eight 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 eight chapters already available. The current material covers all the basics, including higher-order functions and 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.



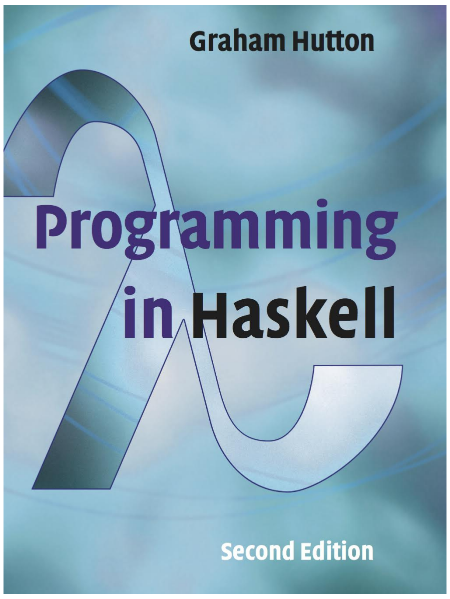
Further chapters will be made available as we complete them.

Further reading

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

2.4 Programming in Haskell - 2nd Edition

Report by:	Graham Hutton
Status:	published September 2016



Overview

Haskell is a purely functional language that allows programmers to rapidly develop software that is clear, concise and correct. This book is aimed at a broad spectrum of readers who are interested in learning the language, including professional programmers, university students and high-school students. However, no programming experience is required or assumed, and all concepts are explained from first principles with the aid of carefully chosen examples and exercises. Most of the material in the book should be accessible to anyone over the age of around sixteen with a reasonable aptitude for scientific ideas.

Structure

The book is divided into two parts. Part I introduces the basic concepts of pure programming in Haskell and is structured around the core features of the language, such as types, functions, list comprehensions, recursion and higher-order functions. Part II covers impure programming and a range of more advanced topics, such as monads, parsing, foldable types, lazy evaluation and reasoning about programs. The book contains many extended programming examples, and each chapter includes suggestions for further reading and a series of exercises. The appendices provide solutions to selected exercises, and a summary of some of the most commonly used definitions from the Haskell standard prelude.

What's New

The book is an extensively revised and expanded version of the first edition. It has been extended with new chapters that cover more advanced aspects of Haskell, new examples and exercises to further reinforce the concepts being introduced, and solutions to selected exercises. The remaining material has been completely reworked in response to changes in the language and feedback from readers. The new edition uses the Glasgow Haskell Compiler (GHC), and is fully compatible with the latest version of the language, including recent changes concerning applicative, monadic, foldable and traversable types.

Further reading

http://www.cs.nott.ac.uk/~pszgmh/pih.html

2.5 Haskell Programming from first principles, a book for all

Report by:	Chris Allen
Participants:	Julie Moronuki
Status:	Content complete, in final editing

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 v0.12.0 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 the material. We expect to release the final version of the book this winter.

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

2.6 Haskell MOOC

Report by:	Jeremy Singer
Participants:	Wim Vanderbauwhede
Status:	Third run of a six-week online Haskell course has just completed

The School of Computing Science at the University of Glasgow has partnered with the FutureLearn platform to deliver a six week massive open online course (MOOC) entitled *Functional Programming in Haskell*. The course goes through the basics of the Haskell language, using short videos, an online REPL, multiple choice quizzes and articles.

The third run of the course began on 1 Apr 2018. Around 1500 people signed up for the course, 70% of whom actively engaged with the materials. The most engaging aspect of the activity is the comradely atmosphere in the discussion forums.

The course is in a steady state, running two times per year in April and September. Visit our site to register your interest.

We are continuously refining the learning materials, based on learner feedback from the course. We presented some initial experiences at the Trends in Functional Programming in Education 2017 conference.

Further reading

- https://www.futurelearn.com/courses/functional-programming-haskell
- https://www.cs.kent.ac.uk/people/staff/sjt/TFPIE2017/TFPIE_2017/Papers/TFPIE_2017_paper_5.pdf

3 Implementations

3.1 The Glasgow Haskell Compiler

Report by:	Ben Gamari
Participants:	the GHC developers
Status:	GHC 8.6

2018 saw GHC's first release under its new accelerated release schedule. GHC 8.4.1 contained improvements in GHC's standard libraries, code generation, and hundreds of bug fixes. Our focus is now turned to the next major release of the 8.0 series, GHC 8.6.1.

Major changes in GHC 8.6

Libraries, source language, and type system

- The new `-XNumericUnderscores` extension allows underscores to be used in numeric literals, improving legibility of longer literals.
- The long-awaited `-XBlockArguments` extensions allows `do` and lambda expressions to be used directly as a function argument, eliminating the need for parentheses or an application operator.
- Possibly: The `-XDerivingVia` extension, a proposed relative of `-XGeneralizedNewtypeDeriving` which allows users to derive typeclasses using a generalized form of `newtype` deriving.
- The `Data.Functor.Contravariant` module from the `contravariant` package has been moved into `base`.

Compiler

- The compiler's core simplifier now performs significantly more varieties of numeric constant folding.
- Incomplete pattern match warnings are now offered for guards in pattern bindings and `MultiWayIf` alternatives.
- A new syntax tree representation based on Trees That Grow. This will make it easier for external users to add their own annotations to the `HsSyn` AST. In future this should allow Shayan Najd to harmonise the GHC and Template Haskell ASTs, and for the `ghc-exactprint` annotations to move into the GHC parsed AST (Shayan Najd and Alan Zimmerman).
- Further improvements in support for cross-compilation (Moritz Angerman)
- Replacement of the `make`-based build system with Hadrian. Hadrian, while being usable in GHC 8.4, should be able to replace `make` in nearly all uses. Moreover, it will have significantly better documentation and support relocatable installation trees, a feature unavailable in the current build system (Andrey Mokhov, Zhen Zhang, Moritz Angerman, Alp Mestanogullari)
- Many, many bug fixes.

Runtime system

Significantly improved Windows support with a new I/O manager, long file path compatibility and dynamic linking support (Tamar Christina).

GHC proposals

Since the launch of the GHC proposals process (https://github.com/ghc-proposals/ghc-proposals), over 128 proposals have been created, 41 have been submitted to the committee and 19 have been accepted. These are:

- OverloadedRecordFields (PR #6)
- Update levity polymorphism (PR #29)
- Make Constraint not apart from Type (PR #32)
- Hex floats (PR #37)
- Allow signatures on pattern synonym constructors (PR #42)
- Explicit foralls proposal (PR #55)
- Overhaul deriving instances for empty data types proposal (PR #63)
- Require namespacing fixity declarations for type names (PR #65)
- Extend `-Wall` with `incomplete-uni-patterns` and `incomplete-record-updates` (PR #71)
- Add small primitive types, like `Int8#`/`Word8#` (PR #74)
- Propose underscores in numeric literals (PR #76)
- Deprecate STM invariant mechanism (PR #77)
- Type-level type applications (PR #80)
- Embrace (`Type :: Type`) (PR #83)
- Allow `do` etc. to be used as function arguments without a $ (PR #90)
- As-pattern synonyms (PR #94)
- Unlifted Newtypes (PR #98)
- Proposal for Source Plugins (PR #107)
- Quantified constraints (PR #109)

At the time of writing, 15 proposals are under active discussion by the community and 13 proposals are under review by the committee.

Looking forward: What's hot

GHC is lucky to have a large number of volunteer contributors.

- Matthías Páll Gissurarson has been adding support for significantly improved diagnostics messages for typed holes.
- Ryan Scott has been busily triaging and fixing bugs on a daily basis, and generally helps to keep things running smoothly.
- Mark Karpov of Tweag I/O has been pushing forward GHC's continuous integration reboot. Using computational resources generously provided by Google X, GHC will be moving its continuous integration infrastructure to CircleCI and Appveyor. This will allow us to more easily produce binary distributions
- Boldizsár Németh has been working on improving GHC's plugin story. GHC currently disables to its recompilation checking when compiling with plugin, dramatically increasing build times in common situations.
- Joachim Breitner has been continuing his work on improving GHC's treatment of "join points".
- Michal Terepeta has been performing a variety of refactoring and optimization in the backend as well as introducing support for sub-word-sized fields.
- Andreas Klebinger has been working on improving various facets of GHC's backend code generator. In the past few weeks alone he has contributed performance optimisations for GHC's C-- pass, improved common subexpression elimination, and added infrastructure for taking advantage of branch likelihoods.
- Tamar Christina has continued his work on making GHC run great on Windows. Recently he has been working to finish up a patchset enabling dynamic linking support on Windows. Tamar is also working on a rework of GHC's Windows IO manager implementation. The new implementation will take full advantage of Windows' asynchronous I/O interfaces and should solve dozens of long-standing tickets.
- In addition to contributing valuable code review and bug triaging, Sebastian Graf has contributed fixes to a variety of issues throughout the compiler, including fixes to demand analysis and improvements to common `Enum` instances.
- Recently Patrick Dougherty dusted off a long-dormant patch making the `ghc-heapview` package a first-class citizen. This package allows Haskell programs to introspect the heap
- Andrey Mokhov, Zhen Zhang, Moritz Angermann, Alp Mestanogullari, Tamar Christina, Patrick Dougherty and Tao He have all been working on the finishing the last mile of the switch to GHC's new Shake-based build system, Hadrian.
- One of the larger projects in the pipeline for 8.6 is Alan Zimmerman and Shayan Najd's refactoring of GHC to use the extensible Trees That Grow AST structure.
- Alan Zimmerman has also been looking to teach GHC's parser to parse incrementally, allowing lower latency reparsing during IDE usage.
- Simon Peyton Jones implemented so-called quantified constraints, which have been on the to-do list for over a decade, and were described in a 2017 Haskell Symposium paper. (http://i.cs.hku.hk/~bruno//papers/hs2017.pdf) A GHC proposal (https://github.com/Gertjan423/ghc-proposals/blob/quantified-constraints/proposals/0000-quantified-constraints.rst) to adopt quantified constraints was agreed, so they will appear in GHC 8.6.

As always, if you are interested in contributing to any facet of GHC, be it the runtime system, type-checker, documentation, simplifier, or anything in between, please come speak to us either on IRC (`#ghc` on `irc.freeenode.net`) or `ghc-devs@haskell.org`. Happy Haskelling!

Further reading

- GHC website: https://haskell.org/ghc/
- GHC users guide: https://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 Specific Platforms

3.3.1 Fedora Haskell SIG

Report by:	Jens Petersen
Participants:	Elliott Sales de Andrade, Robert-André Mauchin
Status:	active

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

For the Fedora 28 release on 1st May, ghc was updated to 8.2.2 and packages to Stackage LTS 10 versions. Also the shared dynamic libraries now live in system libdir. Over 30 new packages were added to Fedora 28. We use the cabal-rpm packaging tool to create and update Haskell packages, and fedora-haskell-tools to rebuild them.

For the next release we plan to update to LTS 11, and also to do some packaging changes to subpackage haddock documentation for libraries.

A Fedora Copr repo is available for ghc-8.4.2.

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-lists: https://lists.fedoraproject.org/archives/list/haskell@lists.fedoraproject.org/ and 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/ghc-8.0.2 and https://copr.fedorainfracloud.org/coprs/petersen/stack
- Fedora Haskell Tools: https://github.com/fedora-haskell/fedora-haskell-tools

3.3.2 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 packages and package a wide range of other useful libraries and programs. At the time of writing, we maintain 1077 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 ("strech") provides GHC 8.0.1. In Debian unstable and testing ("buster", the next release) we ship GHC 8.0.2. GHC 8.2 is staged in "experimental".

Debian users benefit from the Haskell ecosystem on 22 architecture/kernel combinations, including the non-Linux-ports KFreeBSD and Hurd.

Further reading

http://wiki.debian.org/Haskell

3.4 Related Languages and Language Design

3.4.1 hs-to-coq

Report by:	Joachim Breitner
Participants:	Antal Spector-Zabusky, Stephanie Weirich
Status:	working

The `hs-to-coq` tool, written by Antal Spector-Zabusky, Joachim Breitner and Stephanie Weirich, translates a large subset of Haskell into Gallina, the programming langauge of the proof assistant Coq. The translation process is configurable and allows the user to stub out low-level features, to map Haskell types and functions to corresponding existing Coq types and functions and to declare termination arguments. It has been used to verify a large subset of Haskell's `containers` library.

Further reading

- https://github.com/antalsz/hs-to-coq/
- https://arxiv.org/abs/1711.09286 paper introducing `hs-to-coq`
- https://arxiv.org/abs/1803.06960 paper discussing the verification of `containers`

3.4.2 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 value as a platform for research and experiments in dependently typed programming.

Release of Agda 2.5.4 is planned for early summer 2018 with a number of new features:

◦ `do`-notation
◦ Compile-time call-by-need evaluation
◦ Builtin 64-bit words
◦ Improved performance of compiled code

Further reading

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

3.4.3 Disciple

Report by:	Ben Lippmeier
Participants:	Ben Lippmeier, Jacob Stanley
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 v0.5.1 was released in late October, and is in "working alpha" state. The main new features are:

◦ Copying garbage collection using the LLVM shadow stack.
◦ Implicit parameters, which support Haskell-like ad-hoc overloading using dictionaries.
◦ Floating point primitives.
◦ Travis continuous integration for the GitHub site.
◦ A new Sphinx based user guide and homepage.

We are currently working on a new indexed binary format for interface files, as re-parsing interface files is currently a bottleneck. The file format is to be provided by the Shimmer project, which has been split out into a separate repo.

Further reading

◦ http://disciple.ouroborus.net
◦ https://github.com/DDCSF/shimmer

4 Libraries, Tools, Applications, Projects

4.1 Language Extensions and Related Projects

4.1.1 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.

Building on this prior work, I have written my dissertation on incorporating proper dependent types in Haskell [2]. I have yet to have the time to start genuine work on the implementation, but I plan to do so starting summer 2017.

Here is a sneak preview of what will be possible with dependent types, although much more is possible, too!

```
data Vec :: ∗ → Integer → ∗ where
  Nil   :: Vec a 0
  (:::) :: a → Vec a n → Vec a (1 '+ n)
replicate :: π n. ∀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]: *Dependent Types in Haskell: Theory and Practice*, by Richard A. Eisenberg. PhD Thesis, 2015.
  https://github.com/goldfirere/thesis/tree/master/built

4.1.2 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 new talk from ZuriHack 2016 is available on Youtube. The most interesting upcoming feature is probably type-level metadata, making use of the fact that GHC 8 now offers type-level metadata for the built-in generics. While the feature is in principle implemented, there are still a few open questions about what representation would be most convenient to work with in practice. Help or opinions are welcome!

Further reading

- `generics-sop` package:
  https://hackage.haskell.org/package/generics-sop/
- Tutorial (summer school lecture notes):
  https://github.com/kosmikus/SSGEP/
- ZuriHac 2016 talk:
  https://www.youtube.com/watch?v=sQxH349HOik
- WGP 2014 talk:
  https://www.youtube.com/watch?v=jzgfM6NFE3Y
- Paper:
  http://www.andres-loeh.de/TrueSumsOfProducts/

4.1.3 Supermonads

Report by:	Jan Bracker
Participants:	Jan Bracker and Henrik Nilsson
Status:	Experimental fully working version

The supermonad package provides a unified way to represent different monadic and applicative notions. In other words, it provides a way to use standard and generalized monads and applicative functors (with additional indices or constraints) without having to manually disambiguate which notion is referred to in every context. This allows the reuse of code, such as standard library functions, across all of the notions.

To achieve this, the library splits the monad and applicative type classes such that they are general enough to allow instances for all of the generalized notions and then aids constraint checking through a GHC plugin to ensure that everything type checks properly. Due to the plugin the library can only be used with GHC.

If you are interested in using the library, we have a few examples of different size in the repository to show how it can be utilized. The generated Haddock documentation also has full coverage and can be seen on the libraries Hackage page.

The project had its first release shortly before ICFP and the Haskell Symposium 2016. Since then we have added support for applicative functors in addition to monads.

We are working on a comprehensive paper that covers all aspects of the project and its theoretical foundations. The paper is submitted to the Journal of Functional Programming.

If you are interested in contributing, found a bug or have a suggestion to improve the project we are happy to hear from you in person, by email or over the projects bug tracker on GitHub.

Further reading

- Hackage:
  http://hackage.haskell.org/package/supermonad
- Repository:
  https://github.com/jbracker/supermonad
- Paper:
  https://jbracker.de/publications/2016-BrackerNilsson-Supermonads.pdf
- Comprehensive JFP Paper:
  https://jbracker.de/publications/2017-BrackerNilsson-SupermonadsAndSuperapplicatives-UnderConsideration.pdf
- Bug-Tracker:
  https://github.com/jbracker/supermonad/issues
- Haskell Symposium presentation:
  https://youtu.be/HRofw58sySw

4.1.4 Reifying type families

Report by:	Gabor Greif
Status:	experimental, comments welcome

The outcome of the compile-time evaluation of type families is currently inscrutable to the running program. `tyfam-witnesses` is a new minimal library that utilises Template Haskell to obtain the necessary artifacts for running the clauses of *closed* type families at execution time. By pattern matching on the outcome all the type-level equalities can be recovered.

For each closed type family in a series of declarations, `witnesses` adds a GADT mirroring its clauses, and a reification function that *runs* it given indexed `TypeReps`. Here is a usage example:

```
type family Elim v f where
  Elim v (v -> c) = c
  Elim v (d -> c) = d -> Elim v c
}
```

gets accompanied with

```
data ElimRefl v f where
  Elim0 :: Elim v (v -> d) ~ d
        => ElimRefl v (v -> d)
  Elim1 :: Elim v (c -> d) ~ (c -> Elim a b)
        => ElimRefl v (c -> d)
```

and a *runner* (or *reifier*)

```
reify_Elim :: TypeRep a -> TypeRep b
           => Maybe (ElimRefl a b)
```

Pattern matching on the result of the latter guides the GHC type checker and allows writing recursive functions that evaluate to an `ElimRefl v f`, which would otherwise get stuck.

The library has been introduced at the Regensburg Haskell Meetup (→ 6.8) and other conferences in Oct. 2017.

You can find it on hackage, grab it with `cabal install tyfam-witnesses`, be reminded however, that GHC v8.2 is a prerequisite for its usage.

I am interested in possible further uses and am waiting for encouragement in resolving the two remaining restrictions.

Further reading

- https://hackage.haskell.org/package/tyfam-witnesses
- https://skillsmatter.com/skillscasts/10947-lightning-talk-engage-clutch-shift-gear-rofl

4.2 Build Tools and Related Projects

4.2.1 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 recently produced new point releases of Cabal/`cabal-install` from the 1.24 branch. Among other things, Cabal 1.24.2.0 includes a fix necessary to make soon-to-be-released GHC 8.0.2 work on macOS Sierra.

Almost 1500 commits were made to the `master` branch by 53 different contributors since the 1.24 release. Among the highlights are:

- Convenience, or internal libraries – named libraries that are only intended for use inside the package. A common use case is sharing code between the test suite and the benchmark suite without exposing it to the users of the package.
- Support for foreign libraries, which are Haskell libraries intended to be used by foreign languages like C. Foreign libraries only work with GHC 7.8 and later.
- Initial support for building Backpack packages. Backpack is an exciting new project adding an ML-style module system to Haskell, but on the package level. See here and here for a more thorough introduction to Backpack.
- `./Setup configure` now accepts an argument specifying the component to be configured. This is mainly an internal change, but it means that `cabal-install` can now perform component-level parallel builds (among other things).
- A lot of improvements in the `new-build` feature (a.k.a. nix-style local builds). Git `HEAD` version of `cabal-install` is now recommended if you use `new-build`. For an introduction to `new-build`, see this chapter of the manual.
- Special support for the Nix package manager in `cabal-install`. See here for more details.
- `cabal upload` now uploads a package candidate by default. Use `cabal upload --publish` to upload a final version. `cabal upload --check` has been removed in favour of package candidates.
- An `--index-state` flag for requesting a specific version of the package index.
- New `cabal reconfigure` command, which re-runs `configure` with most recently used flags.
- New `autogen-modules` field for modules built automatically (like `Paths_PACKAGENAME`).
- New version range operator `^>=`, which is equivalent to `>=` intersected with an automatically-inferred major version bound. For example, `^>= 2.0.3` is equivalent to `>= 2.0.3 && < 2.1`.
- An `--allow-older` flag, dual to `--allow-newer`.
- New Parsec-based parser for `.cabal` files has been merged, but not enabled by default yet.
- The manual has been converted to reST/Sphinx format, improved and expanded.
- Hackage Security has been enabled by default.
- A lot of bug fixes and performance improvements.

Looking Forward

The next Cabal/`cabal-install` versions will be released either in early 2017, or simultaneously with GHC 8.2 (April/May 2017). Our main focus at this stage is getting the `new-build` feature to the state where it can be enabled by default, but there are many other areas of Cabal that need work.

We would like to encourage people considering contributing to take a look at the bug tracker on GitHub and the Wiki, 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

4.2.2 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/

4.2.3 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. Five 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 (→ 4.2.2). 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, Jens Petersen, Luke Murphy, Chris Dornan, and Mihai Maruseac.

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.

Since the last HCAR, we have moved Stackage Nightly to GHC 8.4.1, as well as released LTS 10 and 11 based on GHC 8.2.2.

4.2.4 Stackgo

Report by:	Sibi Prabakaran
Status:	active

A browser plugin (currently supported for Firefox/Google Chrome) to automatically redirect Haddock documentation on Hackage to corresponding Stackage pages, when the request is via search engines like Google/Bing etc. For the case where the package hasn't been added yet to Stackage, no redirect will be made and the Hackage documentation will be available. This plugin also tries to guess when the user would want to go to a Hackage page instead of the Stackage one and tries to do the right thing there.

Further reading

- https://github.com/psibi/stackgo
- https://addons.mozilla.org/en-US/firefox/addon/stackgo
- https://chrome.google.com/webstore/detail/ojjalokgookadeklnffglgbnlbaiackn

4.2.5 pier

Report by:	Judah Jacobson
Status:	experimental

Pier is a command-line tool for building Haskell projects. It is similar in purpose to Stack (→ 4.2.2); it uses `*.cabal` files to configure individual packages, a top-level YAML file to configure the whole project, and Stackage (→ 4.2.3) to get consistent sets of package dependencies. However, Pier attempts to address some of Stack's limitations by exploring a different approach:

- Pier invokes tools such as `ghc` directly, implementing the fine-grained Haskell build logic from (nearly) scratch. In contrast, Stack relies on a separate framework to implement most of its build steps (i.e.,

Cabal-the-library's `Distribution.Simple`), giving it a more coarse control over the build.
- Pier layers its Haskell-specific logic on top of a general-purpose library for hermetic, parallel builds and dependency tracking. That library is itself implemented using Shake, and motivated by tools such as Nix and Bazel. In contrast, Stack's build and dependency logic is more specific to Haskell projects.

Interestingly, Stack originally did depend on Shake, but stopped using it early on. For more information, see write-ups by authors of Stack and Shake.)

Pier is still experimental, but is already able to build most the packages in Stackage (specifically, 90% of the more than 2600 packages in lts-10.3) as well as itself (i.e., `pier build pier`). Notably, packages with custom Setup scripts are not yet supported.

Future plans include: adding more commands such as `pier repl`; improving the usability around the output file store (for example, garbage collection); and exposing the internal library that Pier uses for hermetic build steps and immutable outputs.

Further reading

- https://github.com/judah/pier
- https://hackage.haskell.org/package/pier

4.2.6 Packcheck: Universal CI testing for Haskell packages

Report by:	Harendra Kumar
Status:	Working

packcheck uniformly, consistently builds and comprehensively sanity tests a Haskell package across build tools (stack/cabal) and across all platforms (Linux/MacOS/Windows). You do not need to be familiar with any of the build tools to use it.

packcheck provides a universal CI/build script *packcheck.sh* and config files designed such that you can just copy over the *.travis.yml* and *appveyor.yml* files provided with packcheck to your package repository and your package is CI ready. You can replicate the same testing on your local machine, just copy *packcheck.sh* to your local machine, put it in your PATH, and run it from your package directory:

```
$ packcheck.sh stack
$ packcheck.sh cabal
$ packcheck.sh cabal-new
```

Further reading

- https://github.com/harendra-kumar/packcheck
- https://hackage.haskell.org/package/packcheck

4.2.7 hsinstall

Report by:	Dino Morelli
Status:	stable, actively developed

This is a utility to install Haskell programs on a system using stack. Although stack does have an `install` command, it only copies binaries. Sometimes more is needed, other files and some directory structure. hsinstall tries to install the binaries, the LICENSE file and also the resources directory if it finds one.

Installations can be performed in one of two directory structures. FHS, or the Filesystem Hierarchy Standard (most UNIX-like systems) and what I call "bundle" which is a portable directory for the app and all of its files. They look like this:

- bundle is sort-of a self-contained structure like this:

```
$PREFIX/
  $PROJECT-$VERSION/
    bin/...
    doc/LICENSE
    resources/...
```

- fhs is the more traditional UNIX structure like this:

```
$PREFIX/
  bin/...
  share/
    $PROJECT-$VERSION/
      doc/LICENSE
      resources/...
```

There are two parts to hsinstall that are intended to work together. The first part is a Haskell shell script, `util/install.hs`. Take a copy of this script and check it into a project you're working on. This will be your installation script. Running the script with the `-help` switch will explain the options. Near the top of the script are default values for these options that should be tuned to what your project needs.

The other part of hsinstall is a library. The install script will try to install a `resources` directory if it finds one. the HSInstall library can then be used in your code to locate the resources at runtime.

Note that you only need the library if your software has data files it needs to locate at runtime in the installation directories. Many programs don't have this requirement and can ignore the library altogether.

Source code is available on darcshub, Hackage and Stackage

Further reading

- hsinstall on darcshub
  http://hub.darcs.net/dino/hsinstall
- hsinstall on Hackage
  https://hackage.haskell.org/package/hsinstall
- hsinstall on Stackage
  https://www.stackage.org/package/hsinstall

4.2.8 yesod-rest

Report by:	Sibi Prabakaran
Status:	active

A Yesod scaffolding site with Postgres backend. It provides a JSON API backend as a separate subsite. The primary purpose of this repository is to use Yesod as a API server backend and do the frontend development using a tool like React or Angular. The current code includes a basic example using React and Babel which is bundled finally by webpack and added in the handler `getHomeR` in a type safe manner.

The future work is to make it compatible with yesod-1.6 and then integrate it as part of yesod-scaffold and make it as part of `stack template`.

Further reading

- https://github.com/psibi/yesod-rest
- https://github.com/yesodweb/yesod-scaffold/issues/136

4.2.9 Haskell Cloud

Report by:	Gideon Sireling

Haskell Cloud is a Source-to-Image builder for building Haskell source into a runnable Docker image. It can be used directly with s2i, or deployed on OpenShift.

Using the Haskell Cloud builder, existing Haskell projects can be uploaded, built, and run from the cloud with minimal changes. A choice of pre-installed frameworks is available - see the Wiki for details.

Further reading

- https://bitbucket.org/accursoft/haskell-cloud

4.3 Repository Management

4.3.1 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.

Darcs 2.14.0 was released on April 2018. It features a better support of non-ASCII encodings by default, an improved display of patch dependencies, per-file conflict marking, a more efficient annotate command, and better shell completion.

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/2.14
- http://hub.darcs.net

4.3.2 git-annex

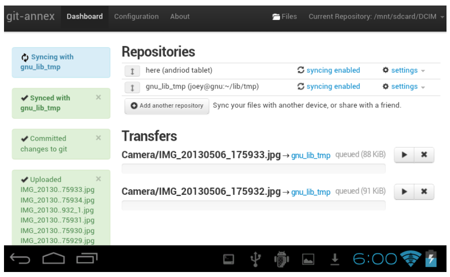
Report by:	Joey Hess
Status:	stable, actively developed

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/

4.4 Debugging and Profiling

4.4.1 Hoed – The Lightweight Algorithmic Debugger for Haskell

Report by:	Maarten Faddegon
Status:	active

Hoed is a tracer and debugger for the programming language Haskell.

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 console.

With Hoed you can list and search observed functions applied to argument values and the result values. Hoed also provides algorithmic debugging. An algorithmic debugger finds defects in programs by systematic search. The programmer directs the search by answering a series of yes/no questions about the correctness of specific function applications and their results. Hoed also allows the use of (QuickCheck-style) properties to answer automatically some of the questions arising during algorithmic debugging, and to replace others by simpler questions.

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)
```

After running the program a computation tree is constructed and the algorithmic debugger is launched in the console.

```
False

=== program terminated ===
Please wait while the computation tree
is constructed...

=== Debug Session ===

hdb> adb
==================================== [0-0/4]
isOdd 3  = False
?
right  Judge computation statements right
        according to the intended
        behaviour/specification
        of the function.
wrong  Judge computation statements wrong
        according to the intended
        behaviour/specification
        of the function.
==================================== [0-0/4]
isOdd 3  = False
? wrong
==================================== [1-0/4]
isEven 4  = False
? wrong
==================================== [2-0/4]
mod2 4  = 2
? wrong
==================================== [3-0/4]
Fault located! In:
mod2 4  = 2
hdb>
```

To reduce the number of questions the programmer has to answer, we added a new mode Assisted Algorithmic Debugging. 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

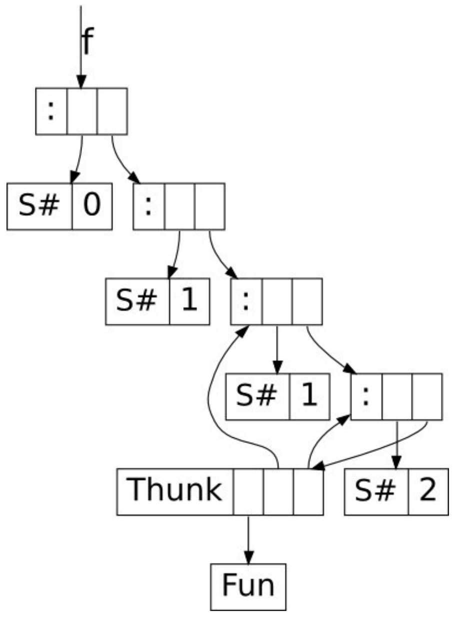
4.4.2 ghc-vis

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

4.4.3 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 (→ 4.4.2) builds on ghc-heap-view.

Since version 0.5.10, ghc-heap-view supports GHC 8.2. There is ongoing work that might merge this functionality into GHC proper.

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
- https://phabricator.haskell.org/D3055 Ongoing work to merge ghc-heap-view into GHC.

4.4.4 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 2.9.4 works with recent versions of 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.

Although Hat is distributed as a cabal package that can be installed with stack, it currently does not support working with stack projects; instead it provides an old-fashioned build tool *hat-make*.

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 (→ 4.4.1) also to Hat.

Further reading

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

4.5 Testing

4.5.1 inspection-testing

Report by:	Joachim Breitner
Status:	working

Carefully crafted Haskell libraries are often set up to trigger a specific cascade of optimization. Stream fusion as found in the `text` and `vector` libraries is a good example, as are generic programming libraries like `generic-lens`. The `inspection-testing` library allows developers to assert that certian transformations indeed happen at compile time, and check these assertions automatically.

For example, the following test file ensures that GHC's optimizer removes the call to `fmap` in `lhs`:

```
{-# LANGUAGE TemplateHaskell #-}
{-# OPTIONS_GHC -O -fplugin
      Test.Inspection.Plugin #-}
import Test.Inspection
import Data.Maybe

lhs, rhs :: (a -> b) -> Maybe a -> Bool
lhs f x = isNothing (fmap f x)

rhs f Nothing = True
rhs f (Just _) = False

inspect $ 'lhs === 'rhs
```

Further reading

- https://github.com/nomeata/inspection-testing
- https://arxiv.org/abs/1803.07130 paper introducing `inspection-testing`

4.5.2 LeanCheck

Report by:	Rudy Braquehais
Participants:	Colin Runciman
Status:	Actively maintained, v0.7.0

LeanCheck is an enumerative property-based testing library with a very small core of only 180 lines of code. Its enumeration is size-bounded so the number of tests is easier to control than with SmallCheck.

It is used like so:

```
> import Test.LeanCheck
> check $ \x y -> x + y == y + (x :: Int)
+++ OK, passed 200 tests.
> check $ \x y -> x - y == y - (x :: Int)
*** Failed! Falsifiable (after 2 tests):
0 1
```

LeanCheck has support for higher-order properties (those taking functions as arguments). For example:

```
> check $ \f p xs -> map f (filter p xs)
>                  == filter p (map f xs
>                               :: [Bool])
*** Failed! Falsifiable (after 20 tests):
\x -> case x of False -> False; True -> False
\x -> case x of False -> False; True -> True
[True]
```

The function `filter` does not commute with `map`.

LeanCheck works on properties whose argument types are instances of the `Listable` typeclass. It is very easy to define `Listable` instances for user-defined types. For example, take "Hutton's Razor":

```
data Expr = Val Int | Add Expr Expr
  deriving (Show, Eq)
```

Its `Listable` instance can be given by

```
instance Listable Expr where
  tiers = cons1 Val \/ cons2 Add
```

or automatically derived using Template Haskell by

```
deriveListable ''Expr
```

LeanCheck is available on Hackage under a BSD3-style license. All you need to do to get it is:

```
$ cabal install leancheck
```

The latest version (v0.7.0) includes functions to compute statistics of `Listable` instances.

Further reading

- https://hackage.haskell.org/package/leancheck
- https://github.com/rudymatela/leancheck
- Chapter 3 of Rudy Braquehais' 2017 PhD Thesis: https://matela.com.br/paper/rudy-phd-thesis.pdf

4.5.3 Extrapolate

Report by:	Rudy Braquehais
Participants:	Colin Runciman
Status:	Actively maintained, v0.3.1

Extrapolate is a property-based testing library capable of reporting generalized counter-examples to properties. Extrapolate works on top of LeanCheck (→ 4.5.2).

Here is an example:

```
> import Test.Extrapolate
> import Data.List (nub)
> check $ \xs -> nub xs == (xs :: [Int])
*** Failed! Falsifiable (after 3 tests):
[0,0]
```

```
Generalization:
x:x:_

Conditional Generalization:
x:xs  when  elem x xs
```

The above property about `nub` not only fails for the list `[0,0]` but also for any list that has repeated elements.

The generalization of failing cases informs the programmer more fully and more immediately what characterizes failures. This information helps the programmer to locate more confidently and more rapidly the causes of failure in their program.

Extrapolate's generalization of counter-examples is similar to SmartCheck's. However, when generalizing, Extrapolate allows for repeated variables and side-conditions.

Extrapolate is available on Hackage under a BSD3-style license. All you need to do to get it is:

```
$ cabal install extrapolate
```

The latest version (v0.3.1) uses Speculate (→ 4.5.4) to avoid testing equivalent conditions improving results and performance.

Further reading

- https://hackage.haskell.org/package/extrapolate
- https://github.com/rudymatela/extrapolate
- IFL 2017 paper about Extrapolate: https://matela.com.br/paper/extrapolate.pdf
- Chapter 6 of Rudy Braquehais' 2017 PhD Thesis: https://matela.com.br/paper/rudy-phd-thesis.pdf

4.5.4 Speculate

Report by:	Rudy Braquehais
Participants:	Colin Runciman
Status:	Actively maintained, v0.3.2

Speculate is a library that uses testing to automatically discover and conjecture properties about Haskell functions. Those properties can contribute to understanding, documentation, validation, design refinement and regression testing.

The following example shows how to use Speculate to discover properties about addition and multiplication:

```
> import Test.Speculate
> speculate args
>   { constants = [ constant "+" (+)
>                 , constant "*" (*) ] }
      x + y == y + x
      x * y == y * x
(x + y) + z == x + (y + z)
(x * y) * z == x * (y * z)
(x + x) * y == x * (y + y)

x <= x * x
```

Speculate can even discover properties about higher-order functions. For example, it discovers the following properties about `map`, `id` and `(.)` (cf. `eg/fun.hs`):

```
           id x == x
        f (g x) == (f . g) x
      map id xs == xs
map (f . g) xs == map f (map g xs)
         f . id == f
         id . f == f
   (f . g) . h == f . (g . h)
```

Speculate is similar to QuickSpec, but uses a different algorithm to produce *inequalities* and *conditional equations*. See the documentation for further details and examples.

The latest version (v0.3.2) includes significant performance improvements. It also includes improvements in documentation, examples and API.

Speculate is available on Hackage under a BSD3-style license. All you need to do to get it is:

```
$ cabal install speculate
```

Further reading

- https://hackage.haskell.org/package/speculate
- https://github.com/rudymatela/speculate
- The Haskell Symposium 2017 paper about Speculate: https://matela.com.br/paper/speculate.pdf
- Chapter 5 of Rudy Braquehais' 2017 PhD Thesis: https://matela.com.br/paper/rudy-phd-thesis.pdf

4.5.5 TorXakis

Report by:	Damian Nadales
Status:	active development

More often than not, testing software consumes a large portion of the development budget, however we frequently see cases where unit and integration tests fail to uncover critical errors that appear once the software is deployed. Most testing techniques revolve around specifying a collection of execution sequences that check the expected against the actual behavior. A problem with this is that the number of possible execution sequences is huge, and therefore only a very small portion of these would be covered by test cases that are specified as a sequence of steps. The second problem is that, with the goal of increasing coverage and prevent regression bugs a large number of test cases is written, which eats up the development budget.

Model-based testing is a technique for writing tests, where a model of the system behavior is made a a high-level of abstraction, and then the system-under test is tested against this the expected behavior as specified by the model. Model-based testing relies on different algorithms for generating test cases from models, which allows to achieve a much higher test coverage than standard testing techniques, while requiring only a fraction of the code.

TorXakis is such a model-based testing tool, that has been used to verify large scale systems in well-know high tech companies. This tool is entirely written in Haskell, and its code is available on Github under a BSD3 license.

Since July last year, a lot of effort was put into taking TorXakis from a prototype to an industrial grade tool. Some of the improvements made include:

- setup of continuous integration (Windows/Linux), including hlint quality metrics via code climate.
- release of macOS and Windows installers.
- addition of integration tests and benchmarks.
- improvements in performance.
- architectural simplifications.

In addition, there is ongoing work in a new command line interface and a new compiler for the TorXakis language.

A year is almost gone, and there are a lot of interesting challenges ahead to make TorXakis a tool that can be used in production, so we welcome the contributions of anybody interested in the topic.

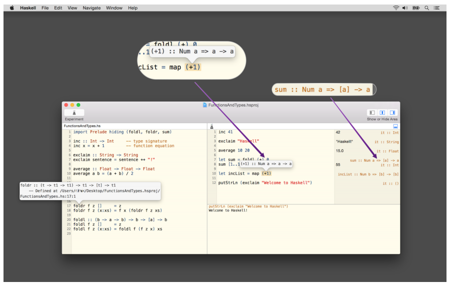
Further reading

https://github.com/TorXakis/TorXakis/

4.6 Development Tools and Editors

4.6.1 Haskell for Mac

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



```
= foldl (+) 0
..1 (+1) :: Num a => a -> a
cList = map (+1)
```

```
sum :: Num a => [a] -> a
```

```
import Prelude hiding (foldl, foldr, sum)

inc :: Int -> Int       -- type signature
inc x = x + 1           -- function equation

exclaim :: String -> String
exclaim sentence = sentence ++ "!"

average :: Float -> Float -> Float
average a b = (a + b) / 2

foldr :: (t -> t1 -> t1) -> t1 -> [t] -> t1
    -- Defined at /Users/[illegible]/Desktop/FunctionsAndTypes.hsproj/
FunctionsAndTypes.hs:17:1

foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

```
inc 41                                   42          it :: Int

exclaim "Haskell"                        "Haskell!"  it :: String

average 10 20                            15.0        it :: Float

let sum = foldl (+) 0                    sum :: Num a => [a] -> a
sum [1..1 (+1) :: Num a => a -> a        55          it :: Int

let incList = map (+1)                   incList :: Num b => [b] -> [b]

putStrLn (exclaim "Welcome to Haskell")              it :: ()
```

```
putStrLn (exclaim "Welcome to Haskell")
Welcome to Haskell!
```

Haskell for Mac is an easy-to-use, innovative programming environment and learning platform 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.

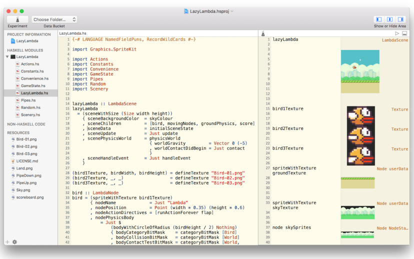
Integrated environment. Haskell for Mac integrates everything needed to start writing Haskell code, including an editor with syntax highlighting and smart identifier completion. Haskell for Mac creates Haskell projects based on standard Cabal specifications for compatibility with the rest of the Haskell 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 & games. 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.

The screenshot below is from the development of a Flappy Bird clone in Haskell. Watch the Haskell for Mac developer live code Flappy Bird in Haskell in 20min at the end of the Compose :: Melbourne 2016 keynote at https://speakerdeck.com/mchakravarty/playing-with-graphics-and-animations-in-haskell. You can find more information about writing games in

Haskell in this blog post: http://blog.haskellformac.com/blog/writing-games-in-haskell-with-spritekit.



```
{-# LANGUAGE NamedFieldPuns, RecordWildCards #-}

import Graphics.SpriteKit

import Actions
import Constants
import Convenience
import GameState
import Pipes
import Random
import Scenery


lazyLambda :: LambdaScene
lazyLambda
  = (sceneWithSize (Size width height))
    { sceneBackgroundColor  = skyColour
    , sceneChildren         = [bird, movingNodes, groundPhysics, score]
    , sceneData             = initialSceneState
    , sceneUpdate           = Just update
    , scenePhysicsWorld     = physicsWorld
                              { worldGravity         = Vector 0 (-5)
                              , worldContactDidBegin = Just contact
                              }
    , sceneHandleEvent      = Just handleEvent
    }

(bird1Texture, birdWidth, birdHeight) = defineTexture "Bird-01.png"
(bird2Texture, _, _)                  = defineTexture "Bird-02.png"
(bird3Texture, _, _)                  = defineTexture "Bird-03.png"

bird :: LambdaNode
bird = (spriteWithTexture bird1Texture)
       { nodeName             = Just "Lambda"
       , nodePosition         = Point (width * 0.35) (height * 0.6)
       , nodeActionDirectives = [runActionForever flap]
       , nodePhysicsBody
           = Just $
               (bodyWithCircleOfRadius (birdHeight / 2) Nothing)
               { bodyCategoryBitMask    = categoryBitMask [Bird]
               , bodyCollisionBitMask   = categoryBitMask [World]
               , bodyContactTestBitMask = categoryBitMask [World,
```

```
lazyLambda                          LambdaScene

bird1Texture                        Texture

bird2Texture                        Texture

bird3Texture                        Texture

spriteWithTexture                   Node userData
groundTexture

spriteWithTexture                   Node userData
skyTexture

node skySprites                     Node NodeSta...
```

Haskell for Mac has recently gained auto-completion of identifiers, taking into account the current module's imports. It now also features a graphical package installer for LTS Haskell and support for GHC 8. Moreover, a new type class, `Presentable`, enables custom rendering of user-defined data types using images, HTML, and even animations.

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.

Further reading

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

4.6.2 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. Work has started on a Language Server Protocol transport, for use in VS Code. `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://github.com/Microsoft/language-server-protocol
- 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/

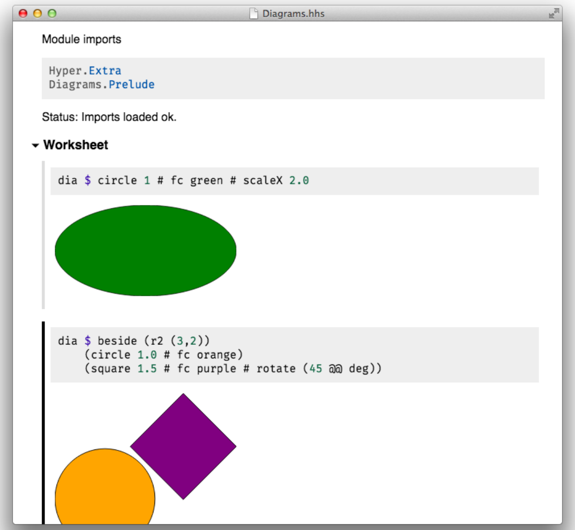
4.6.3 HyperHaskell – The strongly hyped Haskell interpreter

Report by:	Heinrich Apfelmus
Status:	available, active development

HyperHaskell is a graphical Haskell interpreter, not unlike GHCi, but hopefully more awesome. You use worksheets to enter expressions and evaluate them. Results are displayed graphically using HTML.

HyperHaskell is intended to be *easy to install*. It is cross-platform and should run on Linux, Mac and Windows. Internally, it uses the GHC API to interpret Haskell programs, and the graphical front-end is built on the Electron framework. *HyperHaskell* is open source.

HyperHaskell's main attraction is a `Display` class that supersedes the good old `Show` class. The result looks like this:



```
Hyper.Extra
Diagrams.Prelude
```

Status: Imports loaded ok.

```
dia $ circle 1 # fc green # scaleX 2.0
```

```
dia $ beside (r2 (3,2))
    (circle 1.0 # fc orange)
    (square 1.5 # fc purple # rotate (45 @@ deg))
```

Status

HyperHaskell is currently *Level α*. The latest stable release is `0.1.0.2`. Compared to the previous report, no new release has been made, but basic features are working. It is now possible to interpret statements in the `IO` monad and to bind variables, greatly enhancing the usefulness of the interpreter.

Support for the Nix package manager has been implemented by Rodney Lorrimar. I am looking for help in setting up binary releases on the Windows platform!

Future development

Programming a computer usually involves writing a program text in a particular language, a "verbal" activity. But computers can also be instructed by gestures, say, a mouse click, which is a "nonverbal" activity. The long term goal of *HyperHaskell* is to blur the lines between programming "verbally" and "nonverbally" in Haskell. This begins with an interpreter that has graphical representations for values, but also includes editing a program text while it's running ("live coding") and interactive representations of values (e.g. "tangible values"). This territory is still largely uncharted from a purely functional perspective, probably due to a lack of easily installed graphical facilities. It is my hope that *HyperHaskell* may provide a common ground for exploration and experimentation in this direction, in particular by offering the `Display` class which may, perhaps one day, replace our good old `Show` class.

A simple form of live coding is planned for *Level β*, and I am experimenting with interactive music programming.

Further reading

- Project homepage and downloads:
  https://github.com/HeinrichApfelmus/hyper-haskell

4.6.4 CodeWorld

Report by:	Chris Smith
Status:	actively used

CodeWorld is a web-based educational programming environment using Haskell, and appropriate for all ages. It provides a simple mathematical model for geometric figures, animations, and interactive and multi-player games. The language scales between a graphical block-based language for primary students, a simplified variant of Haskell, and the full-fledged Haskell language for older students and universities. In addition to the tools, CodeWorld also provides learning resources for teachers and independent learners. CodeWorld is actively used for Haskell programming classes and activities, by universities, primary and secondary schools, and non-profit organizations and programs.

Features

- A cloud-based programming environment available from anywhere, to write and run code directly in the browser.
- A full-featured Haskell editor with syntax highlighting, rainbow brackets, formatting, and autocomplete.
- A simple graphics model for composable geometry and animations.
- Integrated debugging tools that intelligently link program output to the lines of code responsible.
- The world's simplest framework for single and multiplayer (networked) games.

Recent changes

In the summer of 2017, CodeWorld hosted four students through the Summer of Haskell program to work on improving debugging tools, error messages, collaborative coding experiences, and exporting projects to video and mobile applications. Some contributions are still being merged. Other recent changes include the addition of a model for simple multi-player networked games and QuickCheck support for the full Haskell mode. Simultaneously, we've been busy developing a packaged curriculum for early secondary students, ages 11-14, using functional programming as a framework for creative mathematics.

Availability

CodeWorld is freely available. The hosted web site at http://code.world is open to the public. Source code for the project is available under the Apache license. Teaching materials and resources are released as they are completed under a Creative Commons license.

Further reading

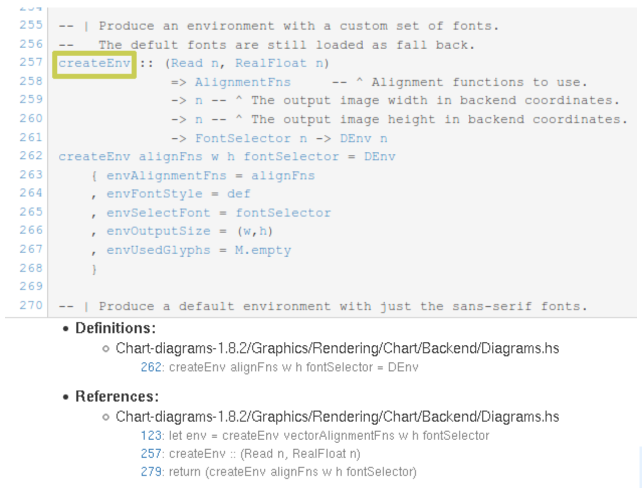
https://github.com/google/codeworld

4.6.5 Haskell Indexer

Report by:	Ivan Krišto
Participants:	Robin Palotai, Kristoffer Søholm
Status:	stable, actively developed

Haskell Indexer is a Kythe extension for working with Haskell source code. Kythe is language-agnostic ecosystem for building tools that work with code. An example is code search with cross-reference support: https://cs.chromium.org/. Haskell Indexer makes it possible to use Kythe-based tools with Haskell. With Haskell Indexer it's possible to list all use-sites of any given function (get reverse-references) and explore the code without any IDE setup.

A portion of GHC and Stackage is indexed and available at http://stuff.codereview.me/.



```
255 -- | Produce an environment with a custom set of fonts.
256 --   The defult fonts are still loaded as fall back.
257 createEnv :: (Read n, RealFloat n)
258                => AlignmentFns      -- ^ Alignment functions to use.
259                -> n -- ^ The output image width in backend coordinates.
260                -> n -- ^ The output image height in backend coordinates.
261                -> FontSelector n -> DEnv n
262 createEnv alignFns w h fontSelector = DEnv
263     { envAlignmentFns = alignFns
264     , envFontStyle = def
265     , envSelectFont = fontSelector
266     , envOutputSize = (w,h)
267     , envUsedGlyphs = M.empty
268     }
269
270 -- | Produce a default environment with just the sans-serif fonts.
```

- **Definitions:**
  - Chart-diagrams-1.8.2/Graphics/Rendering/Chart/Backend/Diagrams.hs
    262: createEnv alignFns w h fontSelector = DEnv
- **References:**
  - Chart-diagrams-1.8.2/Graphics/Rendering/Chart/Backend/Diagrams.hs
    123: let env = createEnv vectorAlignmentFns w h fontSelector
    257: createEnv :: (Read n, RealFloat n)
    279: return (createEnv alignFns w h fontSelector)

Haskell Indexer is in active use and development. As of today, it's possible to get it from GitHub – the Hackage release is work in progress. The future plans include improving the cross-reference support, adding cross-language linking with C and better handling of Template Haskell.

Further reading

- https://github.com/google/haskell-indexer
- https://kythe.io/

4.6.6 Brittany

Report by:	Lennart Spitzner
Status:	work in progress

Brittany is a Haskell source code formatting tool that tries to produce consistent, neat, and concise layouting while preserving comments and user-provided formatting (e.g. newlines). In particular brittany..

- is highly configurable (powerful, alas sparsely documented);
- retains newlines and comments unmodified (to the degree possible when code around them gets reformatted);
- makes clever use of horizontal space without overflowing the column limit (80 or whatever);
- supports horizontal alignment (e.g. different equations/pattern matches in the some function's definition).

The project is not finished in two important aspects:

- Brittany does not know how to format data, class, or instance declarations yet. So it currently only affects a) the module header (exports/imports) b) type signatures c) (function) equations.
- Not all extensions are supported yet. E.g. TemplateHaskell/Quasiquoters are not fully supported, plus some other extensions.

Brittany is based on ghc-exactprint and the ghc parser. It requires ghc-8.*, and is available on Hackage and on Stackage.

The project welcomes contribution for any desired, yet missing features.

The design/implementation of this project has been captured in several documents you can find in the project's repository.

Further reading

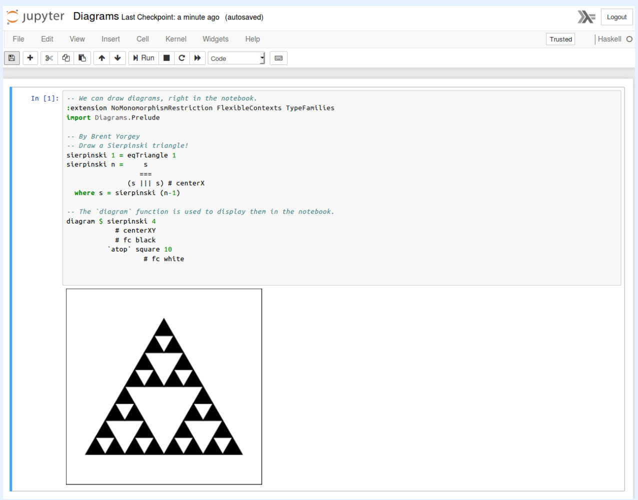
- https://github.com/lspitzner/brittany

4.6.7 IHaskell

Report by:	Vaibhav Sagar
Status:	available, actively maintained

IHaskell is a Haskell kernel for the Jupyter project, and it is usable through various Jupyter frontends, such as the console and the notebook. It is an interactive shell similar to GHCi, and it provides features such as syntax highlighting, autocompletion, multi-line input cells, integrated documentation, rich output visualisation, and more. It integrates with HLint to provide style and formatting suggestions, and notebooks can be exported as Literate Haskell or Markdown.

An example IHaskell notebook looks like this:


jupyter Diagrams Last Checkpoint: a minute ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Haskell
Run
Code
In [1]:

```
-- We can draw diagrams, right in the notebook.
:extension NoMonomorphismRestriction FlexibleContexts TypeFamilies
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 in the notebook.
diagram $ sierpinski 4
            # centerXY
            # fc black
          `atop` square 10
                   # fc white
```

IHaskell currently supports GHC 8.0, 8.2, and 8.4, and older releases have support back to GHC 7.6. It can be installed on Linux and macOS. Windows is currently not supported. Installation via Cabal, Stack, and Nix is supported, and the recently announced mybinder.org can also be used to host IHaskell notebooks.

Further reading

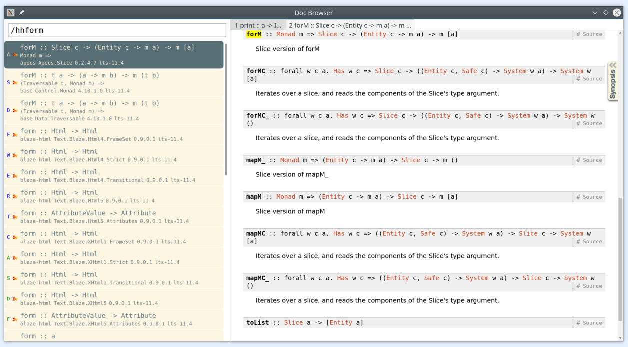
IHaskell is open source, and the project homepage is at https://github.com/gibiansky/IHaskell.

4.6.8 Doc Browser

Report by: Wanqiang Jiang
Status: under development

Doc Browser is an API documentation browser written in Haskell and QML.

It is a native desktop app, with offline supports for Hoogle, DevDocs' docset and Dash's docset.



Doc Browser

/hhform

```
A  forM :: Slice c -> (Entity c -> m a) -> m [a]
   Monad m =>
   apecs Apecs.Slice 0.2.4.7 lts-11.4
S  forM :: t a -> (a -> m b) -> m (t b)
   (Traversable t, Monad m) =>
   base Control.Monad 4.10.1.0 lts-11.4
D  forM :: t a -> (a -> m b) -> m (t b)
   (Traversable t, Monad m) =>
   base Data.Traversable 4.10.1.0 lts-11.4
F  form :: Html -> Html
   blaze-html Text.Blaze.Html4.FrameSet 0.9.0.1 lts-11.4
W  form :: Html -> Html
   blaze-html Text.Blaze.Html4.Strict 0.9.0.1 lts-11.4
E  form :: Html -> Html
   blaze-html Text.Blaze.Html4.Transitional 0.9.0.1 lts-11.4
R  form :: Html -> Html
   blaze-html Text.Blaze.Html5 0.9.0.1 lts-11.4
T  form :: AttributeValue -> Attribute
   blaze-html Text.Blaze.Html5.Attributes 0.9.0.1 lts-11.4
C  form :: Html -> Html
   blaze-html Text.Blaze.XHtml1.FrameSet 0.9.0.1 lts-11.4
A  form :: Html -> Html
   blaze-html Text.Blaze.XHtml1.Strict 0.9.0.1 lts-11.4
S  form :: Html -> Html
   blaze-html Text.Blaze.XHtml1.Transitional 0.9.0.1 lts-11.4
D  form :: Html -> Html
   blaze-html Text.Blaze.XHtml5 0.9.0.1 lts-11.4
F  form :: AttributeValue -> Attribute
   blaze-html Text.Blaze.XHtml5.Attributes 0.9.0.1 lts-11.4
   form :: a
```

1 print :: a -> I... | 2 forM :: Slice c -> (Entity c -> m a) -> m ...

```
forM :: Monad m => Slice c -> (Entity c -> m a) -> m [a]                      # Source
```

Slice version of forM

```
forMC :: forall w c a. Has w c => Slice c -> ((Entity c, Safe c) -> System w a) -> System w
[a]                                                                           # Source
```

Iterates over a slice, and reads the components of the Slice's type argument.

```
forMC_ :: forall w c a. Has w c => Slice c -> ((Entity c, Safe c) -> System w a) -> System w
()                                                                            # Source
```

Iterates over a slice, and reads the components of the Slice's type argument.

```
mapM_ :: Monad m => (Entity c -> m a) -> Slice c -> m ()                      # Source
```

Slice version of mapM_

```
mapM :: Monad m => (Entity c -> m a) -> Slice c -> m [a]                      # Source
```

Slice version of mapM

```
mapMC :: forall w c a. Has w c => ((Entity c, Safe c) -> System w a) -> Slice c -> System w
[a]                                                                           # Source
```

Iterates over a slice, and reads the components of the Slice's type argument.

```
mapMC_ :: forall w c a. Has w c => ((Entity c, Safe c) -> System w a) -> Slice c -> System w
()                                                                            # Source
```

Iterates over a slice, and reads the components of the Slice's type argument.

```
toList :: Slice a -> [Entity a]                                               # Source
```

It is under development, but the GUI is suitable for daily use.

Further reading

- https://github.com/qwfy/doc-browser#readme

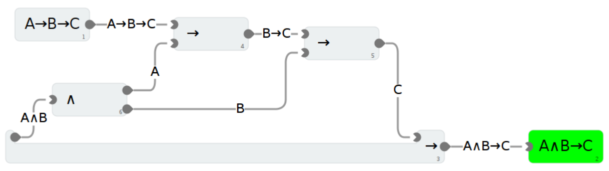
4.7 Formal Systems and Reasoners

4.7.1 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 spaghetti 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

4.7.2 Exference

Report by: Lennart Spitzner
Status: experimental

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.

The current implementation is functional and works well. The most important aspect that still could use improvement is the performance, but it would probably take a slightly improved approach for the core algorithm (and thus a major rewrite of this project) to make significant gains.

The project is actively maintained; apart from occasional bug-fixing and general maintenance/refactoring there are no major new features planned currently.

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

4.8 Education

4.8.1 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.

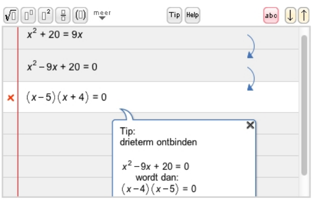
Contact

⟨J.Hage@uu.nl⟩

4.8.2 Interactive Domain Reasoners

Report by:	Bastiaan Heeren
Participants:	Johan Jeuring, Alex Gerdes, Josje Lodder, Hieke Keuning
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), learning environments, and applied games. 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) and the Math-Bridge system. Over the last ten years, we have developed several domain reasoners based on this framework, including reasoners for mathematics, linear algebra, statistics, propositional logic, for learning Haskell (the Ask-Elle programming tutor) and evaluating Haskell expressions and evaluating microcontroller I/O expressions, and for practicing communication skills (the serious game Communicate!, second/right screenshot).



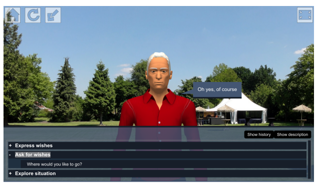
$x^2 + 20 = 9x$

$x^2 - 9x + 20 = 0$

$(x-5)(x+4) = 0$

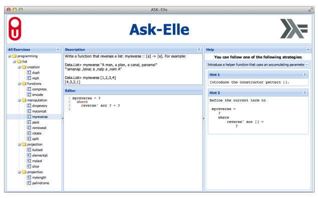
Tip:
drieterm ontbinden

$x^2 - 9x + 20 = 0$
wordt dan:
$(x-4)(x-5) = 0$



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.



Write a function that reverses a list: myreverse :: [a] -> [a]. For example:

```
Data.List> myreverse "A man, a plan, a canal, panama!"
"!amanap ,lanac a ,nalp a ,nam A"

Data.List> myreverse [1,2,3,4]
[4,3,2,1]
```

```
myreverse = ?
  where
    reverse' acc ? = ?
```

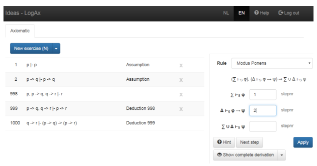
You can follow one of the following strategies:

Introduce a helper function that uses an accumulating paramete

Introduce the constructor pattern [].

Refine the current term to

```
myreverse =
   ?
   where
      reverse' acc [] =
         ?
```



1	p |- p	Assumption	x
2	p -> q |- p -> q	Assumption	x
998	p, p -> q, q -> r |- r		x
999	p -> q, q -> r |- p -> r	Deduction 998	x
1000	q -> r |- (p -> q) -> (p -> r)	Deduction 999	

Rule: Modus Ponens

$(\Sigma \vdash_S \varphi), (\Delta \vdash_S \varphi \to \psi) \Rightarrow \Sigma \cup \Delta \vdash_S \psi$

$\Sigma \vdash_S \varphi$	1	stepnr
$\Delta \vdash_S \varphi \to \psi$	2	stepnr
$\Sigma \cup \Delta \vdash_S \psi$		stepnr

We have started with the Advise-Me project (Automatic Diagnostics with Intermediate Steps in Mathematics Education), which is a Strategic Partnership in EU's Erasmus+ programme. In this project we develop innovative technology for calculating detailed diagnostics in mathematics education, for domains such as 'Numbers' and 'Relationships'. The technology is offered as an open, reusable set of feedback and assessment services. The diagnostic information is calculated automatically based on the analysis of intermediate steps.

We are continuing our research in various directions. We are investigating feedback generation for axiomatic proofs for propositional logic. 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.

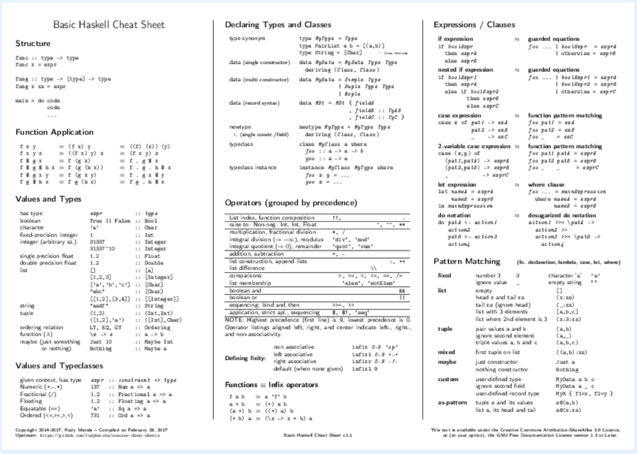
The library for developing domain reasoners with feedback services is available as a Cabal source package. The latest release has support for constructing constraint-based tutors, besides the model-tracing tutors based on problem-solving procedures that are expressed in an extensible domain-specific language. We have written a tutorial on how to make your own domain reasoner with this library.

Further reading

- Hugo Arends, Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. An Intelligent Tutor to Learn the Evaluation of Microcontroller I/O Programming Expressions. Koli Calling Conference on Computing Education Research, 2–9, 2017.
- Bastiaan Heeren and Johan Jeuring. An Extensible Domain-Specific Language for Describing Problem-Solving Procedures. Artificial Intelligence in Education, 77–89, 2017.
- Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and Thomas van Binsbergen. Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. International Journal of Artificial Intelligence in Education, volume 27 (1), 65–100, 2017.
- Josje Lodder, Bastiaan Heeren, and Johan Jeuring. Generating hints and feedback for Hilbert-style axiomatic proofs. SIGCSE '17, 387–392, 2017.
- 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.

4.8.3 Basic Haskell Cheat Sheet

Report by:	Rudy Braquehais
Status:	Actively maintained, v1.1



Basic Haskell Cheat Sheet

Structure

```
func :: type -> type
func x = expr

fung :: type -> [type] -> type
fung x xs = expr

main = do code
          code
          ...
```

Function Application

```
f x y         ≡ (f x) y         ≡ ((f) (x)) (y)
f x y z       ≡ ((f x) y) z     ≡ (f x y) z
f $ g x       ≡ f (g x)         ≡ f . g $ x
f $ g $ h x   ≡ f (g (h x))     ≡ f . g . h $ x
f $ g x y     ≡ f (g x y)       ≡ f . g x $ y
f g $ h x     ≡ f g (h x)       ≡ f g . h $ x
```

Values and Types

has type	expr	:: type
boolean	True || False	:: Bool
character	'a'	:: Char
fixed-precision integer	1	:: Int
integer (arbitrary sz.)	31337	:: Integer
	31337^10	:: Integer
single precision float	1.2	:: Float
double precision float	1.2	:: Double
list	[]	:: [a]
	[1,2,3]	:: [Integer]
	['a','b','c']	:: [Char]
	"abc"	:: [Char]
	[[1,2],[3,4]]	:: [[Integer]]
string	"asdf"	:: String
tuple	(1,2)	:: (Int,Int)
	([1,2],'a')	:: ([Int],Char)
ordering relation	LT, EQ, GT	:: Ordering
function (λ)	\x -> e	:: a -> b
maybe (just something or nothing)	Just 10	:: Maybe Int
	Nothing	:: Maybe a

Values and Typeclasses

given context, has type	expr	:: constraint => type
Numeric (+,-,*)	137	:: Num a => a
Fractional (/)	1.2	:: Fractional a => a
Floating	1.2	:: Floating a => a
Equatable (==)	'a'	:: Eq a => a
Ordered (<=,>=,>,<)	731	:: Ord a => a

Copyright 2014-2017, Rudy Matela – Compiled on February 28, 2017
Upstream: https://github.com/rudymatela/concise-cheat-sheets

Declaring Types and Classes

type synonym	`type MyType = Type`
	`type PairList a b = [(a,b)]`
	`type String = [Char]   -- from Prelude`

data (single constructor)	`data MyData = MyData Type Type`
	`  deriving (Class, Class)`

data (multi constructor)	`data MyData = Simple Type`
	`            | Duple Type Type`
	`            | Nople`

data (record syntax)	`data MDt = MDt { fieldA`
	`              , fieldB :: TyAB`
	`              , fieldC :: TyC }`

newtype (single constr./field)	`newtype MyType = MyType Type`
	`  deriving (Class, Class)`

typeclass	`class MyClass a where`
	`  foo :: a -> a -> b`
	`  goo :: a -> a`

typeclass instance	`instance MyClass MyType where`
	`  foo x y = ...`
	`  goo x = ...`

Operators (grouped by precedence)

List index, function composition	!!, .
raise to: Non-neg. Int, Int, Float	^, ^^, **
multiplication, fractional division	*, /
integral division (⇒ −∞), modulus	`div`, `mod`
integral quotient (⇒ 0), remainder	`quot`, `rem`
addition, subtraction	+, -
list construction, append lists	:, ++
list difference	\\
comparisons:	>, >=, <, <=, ==, /=
list membership	`elem`, `notElem`
boolean and	&&
boolean or	||
sequencing: bind and then	>>=, >>
application, strict apl., sequencing	$, $!, `seq`

NOTE: Highest precedence (first line) is 9, lowest precedence is 0. Operator listings aligned left, right, and center indicate left-, right-, and non-associativity.

Defining fixity:

non associative	infix 0-9 `op`
left associative	infixl 0-9 +-+
right associative	infixr 0-9 -!-
default (when none given)	infixl 9

Functions ≡ Infix operators

```
f a b    ≡ a `f` b
a + b    ≡ (+) a b
(a +) b  ≡ ((+) a) b
(+ b) a  ≡ (\x -> x + b) a
```

Basic Haskell Cheat Sheet v1.1

Expressions / Clauses

```
if expression                 ≈  guarded equations
if boolExpr                      foo ... | boolExpr  = exprA
  then exprA                             | otherwise = exprB
  else exprB
nested if expression          ≈  guarded equations
if boolExpr1                     foo ... | boolExpr1 = exprA
  then exprA                             | boolExpr2 = exprB
  else if boolExpr2                      | otherwise = exprC
         then exprB
         else exprC
case expression               ≈  function pattern matching
case x of pat1 -> exA            foo pat1 = exA
          pat2 -> exB            foo pat2 = exB
          _    -> exC            foo _    = exC
2-variable case expression    ≈  function pattern matching
case (x,y) of                    foo pat1 patA = exprA
  (pat1,patA) -> exprA           foo pat2 patB = exprB
  (pat2,patB) -> exprB           foo _    _    = exprC
  _           -> exprC
let expression                ≈  where clause
let nameA = exprA                foo ... = mainExpression
    nameB = exprB                  where nameA = exprA
in mainExpression                        nameB = exprB
do notation                   ≈  desugarized do notation
do patA <- action1               action1 >>= \patA ->
   action2                         action2 >>
   patB <- action3                 action3 >>= \patB ->
   action4                           action4
```

Pattern Matching (fn. declaration, lambda, case, let, where)

fixed	number 3	3	character 'a'	'a'
	ignore value	_	empty string	""
list	empty	[]		
	head x and tail xs	(x:xs)		
	tail xs (ignore head)	(_:xs)		
	list with 3 elements	[a,b,c]		
	list where 2nd element is 3	(x:3:xs)		
tuple	pair values a and b	(a,b)		
	ignore second element	(a,_)		
	triple values a, b and c	(a,b,c)		
mixed	first tuple on list	((a,b):xs)		
maybe	just constructor	Just a		
	nothing constructor	Nothing		
custom	user-defined type	MyData a b c		
	ignore second field	MyData a _ c		
	user-defined record type	MyR { f1=x, f2=y }		
as-pattern	tuple s and its values	s@(a,b)		
	list a, its head and tail	a@(x:xs)		

This text is available under the Creative Commons Attribution-ShareAlike 3.0 Licence, or (at your option), the GNU Free Documentation License version 1.3 or Later.

The Basic Haskell Cheat Sheet is a reference sheet with reminders of the most common Haskell features and functions condensed in two pages.

It covers a subset of the standard prelude, including examples of:

- declarations & expressions;
- operators & functions;
- list comprehensions;
- GHC invocation & GHCi commands.

It does not aim to teach readers how to use the language, but simply remind readers of syntax, functions or caveats. It can be useful both to newcomers and oldtimers of the Haskell language.

Contributions are welcome, just submit a "pull request" on GitHub. Note the hard limit of two pages: to add something, something else should be removed.

Further reading

- Current version (v1.1): https://github.com/rudymatela/concise-cheat-sheets/releases/download/haskell-v1.1/haskell-cs-1.1.pdf
- Future versions will be posted on: https://github.com/rudymatela/concise-cheat-sheets

4.8.4 DSLsofMath

Report by:	Patrik Jansson
Participants:	Cezar Ionescu, Daniel Heurlin, Daniel Schoepe
Status:	active development

"Domain Specific Languages of Mathematics" is a project at Chalmers and UGOT which lead to a new BSc level course of the same name, including accompanying material for learning and applying classical mathematics (mainly basics of 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 third instance of the course was carried out Jan–March 2018 in Gothenburg and the course material is available on github. The lecture notes have been collected in an informal "book" during the last six months; contributions and ideas are welcome!

Just two example to give a feeling for the contents:

Given the definition of "$h : \mathsf{A} \to \mathsf{B}$ is a homomorphism from $\mathsf{Op} : \mathsf{A} \to \mathsf{A} \to \mathsf{A}$ to $op : \mathsf{B} \to \mathsf{B} \to \mathsf{B}$":

$$\mathsf{H}_2(h, \mathsf{Op}, op) = \forall\ x.\ \forall\ y.\ h\ (\mathsf{Op}\ x\ y) == op\ (h\ x)\ (h\ y)$$

Then $\mathsf{H}_2(exp, (+), (\cdot))$ says that $e^{x+y} == e^x \cdot e^y$ for all x and y.

A somewhat longer example (two tweets):

```
import Prelude hiding (exp, sin, cos)
z = zipWith
c ∫ f = c : z (/) f [1 ..]
exp   = 1 ∫ exp
sin   = 0 ∫ cos
cos   = 1 ∫ (−sin)
instance Num a ⇒ Num [a] where
   (+) = z (+)
   (−) = z (−)
   (∗)  = m
   fromInteger i = fromInteger i : repeat 0
m (x : xs) q@(y : ys) = x ∗ y : (map (x∗) ys) + xs ∗ q
one = 1 :: [Rational]
test = (sin^2 + cos^2) ≃ one
d cs = zipWith (∗) [1 ..] (tail cs)
xs ≃ ys = and (take 10 (z (==) xs ys))
test2 = exp ≃ d exp
```

See also the HCAR entry "Learn you a Physics" (→ 4.8.5) by a group of BSc students.

Further reading

- DSLsofMath (github organisation)
- "Learn you a Physics" BSc project applying DSLsofMath ideas.
- Latest snapshot of the DSLsofMath Lecture Notes (work in progress)
- Exam 2018 with solutions
- TFPIE 2015 paper

4.8.5 Learn You A Physics

Report by:	Erik Sjöström
Participants:	Oskar Lundström, Johan Johansson, Björn Werner
Status:	active development

Learn You A Physics is the result of a BSc project at Chalmers (supervised by P. Jansson) where the goal is to create an introductory learning material for physics aimed at programmers with a basic understanding of Haskell.

It does this by identifying key areas in physics with a well defined scope, for example dimensional analysis or single particle mechanics, and develops a domain specific language around each of these areas.

The implementation of these DSL's are the meat of the learning material with accompanying text to explain every step and how it relates to the physics of that specific area.

The text is written in such a way as to be as non-frightening as possible, and to only require a beginner knowledge in Haskell. Inspiration is taken from Learn you a Haskell for Great Good and the project DSLsofMath (→ 4.8.4) at Chalmers and University of Gothenburg.

The source code and learning material is freely available online.

Further reading

Learn You A Physics

4.9 Text and Markup

4.9.1 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

4.9.2 Fast Unicode Normalization

Report by:	Harendra Kumar
Status:	Working

Unicode strings need to be converted to a normalized form using the Unicode Character Database before they can be compared for equivalence. *unicode-transforms* is a pure Haskell implementation of Unicode normalization. The alternative is the *text-icu* package which provides this functionality as Haskell bindings to the ICU C++ implementation. *unicode-transforms* supports all forms of normalization (NFC/NFD/NFKC/NFKD) and supports the latest version of the Unicode standard (Unicode 9).

One of the goals of *unicode-transforms* was high performance. We have successfully achieved this goal for decompose (NFD/NFKD) forms, achieving performance close to, and in one benchmark even better than the C++ implementation (i.e. the *text-icu* package). Compose (NFC/NFKC) implementation is not yet optimized, and though the performance of compose is decent it is not at par with the C++ implementation. This is still open for anyone looking for a challenge to beat C++.

This library can potentially be integrated with the *text* package allowing us to keep text in a standard normalized form by default, thus freeing the Haskell programmers from worrying about explicit normalization. The library is available on Hackage under BSD3 license.

Further reading

https://github.com/harendra-kumar/unicode-transforms

4.9.3 Automatic type inference from JSON

Report by:	Michał 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 tool now supports Elm output, and plans to support all members of Haskell family of languages.

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

Further reading

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

4.9.4 Ginger

Report by:	Tobias Dammers
Status:	Active development

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. Additionally, some features have been added that are missing in Jinja2, such as lambdas, being able to use macros as functions or filters, 'do' expressions, output indenting constructs, and "script mode", switching Ginger into a syntax that is closer to a unityped scripting language than a template language.

Improvement that haven't made it yet 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. Contributions of any kind are very welcome.

Further reading

- https://ginger.tobiasdammers.nl/
- https://github.com/tdammers/ginger
- http://hackage.haskell.org/package/ginger
- http://jinja2.pocoo.org (the original Jinja2, not my work)

4.10 Web

4.10.1 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 started using Builder of ByteString instead of Builder of blaze-builder. Because they are identical, no backward compatibility issue happens.

Further reading

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

4.10.2 Warp

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

Warp is a high performance, easy to deploy HTTP handler for WAI (→ 4.10.1). Its supports both HTTP/1.1 and HTTP/2.

Since the last HCAR, no major changes were made.

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

4.10.3 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 scalability comes from the amazing GHC compiler and runtime. GHC provides fast code and built-in event-based 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 (→ 4.11.1) — 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 (→ 4.11.1). 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 (→ 4.10.1) 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 and Servant.

Since the last HCAR, we've released version 1.6, the first breaking version of Yesod since September 2014. Most code will continue to work unchanged, but improvements have been added to Yesod's monad transformer implementation, as well as adding some long-awaited cleanups.

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/

4.10.4 Snap Framework

Report by:	Doug Beardsley
Participants:	Gregory Collins, Greg Hale, 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.

Snap continues to see active use in industry and continued support from the development team. 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

- http://snapframework.com
- snap-server test coverage: https://snapframework.github.io/snap-code-coverage/snap-server/hpc-ghc-8.0.1/hpc_index.html
- Snaplet Directory: http://snapframework.com/snaplets
- io-streams: http://hackage.haskell.org/package/io-streams

4.10.5 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

4.10.6 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 with functional dependencies, rank-n types, and row polymorphism with a form of polymorphic labels.

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.

Further reading

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

4.10.7 Sprinkles

Report by:	Tobias Dammers
Status:	Active development

Sprinkles is a "zero programming web development framework", intended for building content-centric server-site websites in a declarative fashion.

As such, Sprinkles sits in a unique position in between static site generators such as Jekyll, Hakyll, etc., and fully dynamic CMS solutions like Ghost, WordPress, etc.

A Sprinkles website consists of a project file in YAML format, a set of templates (using Ginger, a Haskell implementation of the Jinja HTML template language), and static files such as stylesheets, images, and client-side scripts. Data can be loaded from SQL databases, local files, HTTP, or local shell scripts, and Sprinkles supports a wide range of input formats, including JSON, YAML, HTML, Markdown, LaTeX, and even DOCX. The heart of a project file is a list of routes, each specifying a list of backend data sources to query, the output of which then gets bound to a template variable.

Unlike a static site generator, Sprinkles generates its HTML output on the fly, so there is no build step, you just deploy your updated project files to the server, restart the Sprinkles process, and your new site is live. Unlike a classic CMS, however, there is no admin area; instead, you manage your data externally.

This approach has a few advantages, most notably, it makes Sprinkles more secure, because no admin area is exposed that could be exploited to escalate from compromising the public-facing part to gain write access; write access is implemented separately, via separate channels (typically an SSH connection).

Various workflows are possible, e.g.:

- hand-editing markdown files or even Word documents
- putting your data in an SQL database, using a graphical SQL frontend to modify things, and pointing Sprinkles at the database
- using an existing website as your data source, but building a Sprinkles frontend on top
- using github as your data source (the Ginger homepage at https://ginger.tobiasdammers.nl/ does this for the user guide section)
- using something like https://form.io/ to manage your data, and pointing Sprinkles at an API endpoint
- ...

Sprinkles comes with a built-in Warp server, and this is the preferred deployment option, but CGI, SCGI and FastCGI are also supported. A "bake" mode has been added recently that turns Sprinkles into a static site generator, so a Jekyll-style workflow is now also possible.

Further reading

- https://sprinkles.tobiasdammers.nl/
- https://github.com/tdammers/sprinkles
- http://hackage.haskell.org/package/sprinkles

4.10.8 nginx-haskell-module

Report by:	Alexey Radkov
Status:	stable, actively developed

The nginx-haskell-module allows using typed Haskell handlers from within custom Nginx configurations. It provides several Nginx directives that correspond to where Haskell code is applied and what kind of task it solves:

- *haskell_run* — run Haskell handler synchronously and return a string-like result;
- *haskell_run_async* — run Haskell handler asynchronously in the early rewrite Nginx phase and return a string-like result when ready;
- *haskell_content* — run Haskell handler for rendering content;
- *haskell_static_content* — optimized content rendering;
- *haskell_unsafe_content* — optimized (in another way) content rendering;
- *haskell_run_service* — run Haskell handler as a service, meaning that this implements not a request-driven task but rather timer-driven and is directly bound to the Nginx event loop.
- *haskell_service_hook* — run Haskell handler that alters a specific global state as a content handler, that can be used to build custom APIs for managing Haskell services;
- *haskell_service_update_hook* — run Haskell handler in an Nginx worker's main thread after the service has reported a result, that can be used to run *C plugins* related to the service.

Here, "a string-like result" is an umbrella term for various Haskell string-like implementations like String and ByteString, perhaps wrapped inside IO Monad. Typed Haskell handlers give a strong guarantee that underlying code will cause no IO specific side effects thus making sure that Nginx will not stop working due to unpredictably long-running IO code, which is extremely crucial for this popular web server.

On the other hand, when effectful code is desirable, it can be declared with directive *haskell_run_async*, which runs Haskell code asynchronously and won't stop the Nginx world. Moreover, the module provides directive *haskell_run_service* for running custom asynchronous "services" that are not bound to requests: this makes it possible to program interesting solutions like Nginx-specific service discovery (see a reference to an advanced example below). An asynchronous service results are available in request-specific Nginx configuration areas (e.g. location) via an Nginx variable that holds a string, which, on a deeper level, may wrap a typed data (or JSON, or other representations).

Basically, a service results variable is stored in a worker-specific memory, but with directive *haskell_service_var_in_shm* it gets its place in a shared memory and becomes available from all Nginx workers. In conjunction with directive *haskell_service_var_update_callback*, this allows programming custom ad-hoc solutions when a Haskell service's results are needed in another Nginx module. On the Github project page there is an advanced example *dynamicUpstreams* that implements such an approach.

Starting from version 1.4.0 of the module, services that share their results in shared memory become *shared* themselves. This means that only one worker process really runs a shared service, while services on the other workers wait on locks until the active worker exits or dies.

Starting from version 1.8.0 of the module, services can be managed by custom *hooks* declared with directive *haskell_service_hook*. With the hooks, it is possible to build custom APIs for managing services of arbitrary complexity. A similar directive *haskell_service_update_hook* allows for writing *C plugins* that run in a worker's main thread upon every update from the service. To access internal Nginx global data in a plugin, the Haskell module provides several *opaque pointers*, in particular *ngxCyclePtr*, *ngxUpstreamMainConfPtr*, and *ngxCachedTimePtr*.

The module provides other two directives *haskell_var_nocacheable* and *haskell_var_compensate_uri_changes* that make Nginx *error_page* redirections almost Turing-complete. It is achieved by letting them loop without limits on number of cycles (by the second directive) and ensuring that loop-specific variables get updated between iterations (by the first directive). The technique of Turing-complete error_page redirections is used in an advanced example *labeledMediaRouting*.

Further reading

- Project page
- Module ngx-export on Hackage
- An advanced example in a blog article
- A comprehensive tutorial with many examples in PDF

4.10.9 Template Toolkit

Report by:	Dzianis Kabanau
Status:	active, working

Haskell implementation of popular Perl template processing system.

Perl Template Toolkit is mainly used for web development.

This port includes such features as:

- Perl-like variables: scalar, array, hash
- Variables interpolation and autovivification.
- Conditional directives
- Loops and loop controls
- Processing of external templates and internal subtemplate blocks
- Many virtual methods and filters

In the next release it is planned to include support for custom methods, filters and functions in templates.

Further reading

https://hackage.haskell.org/package/template-toolkit

4.11 Databases

4.11.1 Persistent

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

Since the last HCAR, 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 useful 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 for the places where it can be easily 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 (→ 4.10.3) 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

4.11.2 Squeal

Report by:	Eitan Chatav
Status:	experimental, active development

The `squeal-postgresql` package is a library providing a deep embedding of PostgreSQL in Haskell. Squeal makes use of lots of features and ideas in Haskell such as:

- DataKinds
- OverloadedLabels
- Indexed monads
- Generics
- GADTs

The point is not to use them in particularly new ways, but for very practical purposes. DataKinds are used to embed the SQL type and schema system into Haskell. OverloadedLabels are used to refer to tables and columns. Indexed monads are used to track changing schema types. Generics are used to easily marshal data between Haskell product and record types and SQL parameter and row types. GADTs are used for joins, aliased expressions, heterogeneous lists and more.

However, in spite of these perhaps being formidable sounding concepts, it is much easier to learn them when there is a familiar pattern you can use, and that's SQL. I believe that if a Haskeller knows SQL, then Squeal can be a gateway to learning these concepts. That's because Squeal tries to be (mostly) faithful to SQL.

If the Squeal expressions are basically SQL, then why not just use SQL? Squeal adds strong, static typing and decomposability, which helps to develop and maintain an application. Change your schema and the compiler will tell you where you must change your queries and manipulations. Queries and manipulations with common subparts can be factored for reusability. You can even write typed schema migration definitions using Squeal.

Squeal is still experimental and I would love to get some real users and contributors. Send me bug reports and feature requests. There are still many missing features that I have plans to include.

What is new?

Since the last HCAR, Squeal has released version 0.2.1. This is a minor update over version 0.2 which was released late March. The difference between these two versions is solving an issue where alias identifiers could conflict with reserved words in PostgreSQL. To fix the issue, alias identifiers are now quoted. Thanks to Petter Rasmussen for the fix.

The version 0.2 was released March 26th and has the following changes:

- Constraints: Type level table constraints like primary and foreign keys and column constraints like having `DEFAULT`
- Migrations: Support for linear, invertible migrations tracked in an auxiliary table
- Arrays: Support for fixed- and variable-length arrays
- Aliases: Generalized ‘Has‘ constraint
- Pools: Support for pooled connections
- Transactions: Support for transaction control language
- Queries, Manipulations, Definitions: Small and large changes to Squeal’s DSL

Bringing SQL constraints to the type level means that more of the schema is statically known. In version 0.1 column constraints – which boil down to having ‘DEFAULT‘ or not – were at the type level, but they were confusingly named.

```
0.1: 'Optional ('NotNull 'PGInt4)
0.2: 'Def :=> 'NotNull 'PGInt4
0.1: 'Required ('NotNull 'PGInt4)
0.2: 'NoDef :=> 'NotNull 'PGInt4
```

The `:=>` type operator is intended to helpfully connote a constraint relation. It’s also used for table constraints which are new in version 0.2:

```
"emails" :::
  '[ "pk_emails"  ::: 'PrimaryKey '["id"]
   , "fk_user_id" ::: 'ForeignKey '["user_id"]
       "users" '["id"]
   ] :=>
  '[ "id"      :::   'Def :=> 'NotNull 'PGint4
   , "user_id" ::: 'NoDef :=> 'NotNull 'PGint4
   , "email"   ::: 'NoDef :=>    'Null 'PGtext
   ]
```

Another change in the constraint system was the removal of column constraints from query and manipulation results, as result columns don’t support a notion of `DEFAULT`. This necessitates a distinction between `TableType`s which have both column and table constraints and `RelationType`s which have neither.

Migrations are a hot topic and many people have requested this feature. From version 0.2, Squeal adds support for linear, invertible migrations. That is, a migration is a named, invertible, schema-tracking computation:

```
data Migration io schema0 schema1 = Migration
  { name :: Text
    -- ^ The ‘name‘ of a ‘Migration‘.
    -- Each ‘name‘ in a ‘Migration‘ should
    -- be unique.
  , up :: PQ schema0 schema1 io ()
    -- ^ The ‘up‘ instruction of a ‘Migration‘
  , down :: PQ schema1 schema0 io ()
    -- ^ The ‘down‘ instruction of a ‘Migration‘
  }
```

And, `Migration`s can be put together in an “aligned” list:

```
data AlignedList p x0 x1 where
  Done :: AlignedList p x x
  (:>>) :: p x0 x1
        -> AlignedList p x1 x2
        -> AlignedList p x0 x2
```

These aligned lists are free categories and might look familiar from the reflections without remorse technique, which uses their cousins, aligned sequences.

In the context of migration, they allow one to chain new migrations as a schema evolves over time. `Migrations`’ execution is tracked in an auxiliary migrations table. Migration lists can then be run or rewinded.

```
migrateUp
  :: MonadBaseControl IO io
  => AlignedList (Migration io)
     schema0 schema1
     -- ^ migrations to run
  -> PQ
    ("schema_migrations"
      ::: MigrationsTable ': schema0)
    ("schema_migrations"
      ::: MigrationsTable ': schema1)
    io ()

migrateDown
  :: MonadBaseControl IO io
  => AlignedList (Migration io)
     schema0 schema1
     -- ^ migrations to rewind
  -> PQ
    ("schema_migrations"
      ::: MigrationsTable ': schema1)
    ("schema_migrations"
      ::: MigrationsTable ': schema0)
    io ()
```

Regarding aliases, in Squeal 0.1, we had different typeclasses `HasColumn` and `HasTable` to indicate that a table has a column or that a schema has a table. In Squeal 0.2 this has been unified to a single typeclass,

```
class KnownSymbol alias =>
  Has (alias :: Symbol)
    (fields :: [(Symbol,kind)])
    (field :: kind)
  | alias fields -> field where
```

Support for array types has been added to Squeal 0.2 through the `'PGfixarray` and `'PGvararray` `PGType`s. Array values can be constructed using the ‘array‘ function and can be encoded from and decoded to Haskell `Vector`s.

Squeal 0.2 provides a monad transformer `PoolPQ` that’s an instance of `MonadPQ`. The `resource-pool` library is leveraged to provide striped pools of `Connections`. `PoolPQ` should be a drop in replacement for running `Manipulations` and `Querys` with `PQ`.

Squeal 0.2 supports a simple transaction control language. A computation in `MonadPQ` can be called `transactionally` with different levels of isolation. Additionally, a schema changing computation, a data definition, can be run in a transaction. Running a computation in a transaction means that all SQL statements will be rolled back if an exception is encountered.

The above changes required major and minor changes to Squeal DSL functions. Please consult the documentation.

Further reading

- https://hackage.haskell.org/package/squeal-postgresql
- https://github.com/morphismtech/squeal

4.11.3 Haskell Relational Record

Report by:	Kei Hibino
Participants:	Shohei Murayama, Shohei Yasutake, Sho Kuroda and Kazu Yamamoto
Status:	stable, active

Haskell Relational Record (HRR) is open-source libraries which provides a pragmatic embedded domain specific language to generate SQL. It supports a large part of the SQL standard which includes outer joins and aggregations with type safety, and also supports pragmatic but type-unsafe features like placeholder, correlation and direct SQL embedding interfaces. Direct SQL embedding interfaces allow database-system-dependent SQL code fragments.

HRR also provides template-haskell which generates record types and relation types definitions from relational-database schemas. Supported schemes of relational-database engines are PostgreSQL, MySQL, SQLite, IBM DB2, Microsoft SQL Server and OracleSQL.

HRR is publicly developed on github since 2013, and its release is publicly announced to Haskell community in December 2014. HRR has been in use at Asahi Net (Internet Service Provider in Japan) since March 2013, and more than three years of production use demonstrates its stability and usability. HRR is actively developed, and technical support is provided in Haskell-jp (→ 6.13).

Further reading

http://khibino.github.io/haskell-relational-record/

4.11.4 YeshQL

Report by:	Tobias Dammers
Status:	Stable, active development

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
```

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)

4.11.5 DBFunctor: Functional Data Management

Report by:	Nikos Karagiannidis
Status:	Stable, active development

`DBFunctor` is a library for *Functional Data Management* of tabular data. What does this mean?

It means that whenever you have a data analysis/preparation/transformation task and you want to do it with Haskell type-safe code, that you enjoy, love and trust so much, now you can!

Moreover, whenever you have a data set in plain CSV files and you want to write a Haskell application that needs to analyze these data by queries, or apply any data transformations on them; you no longer need to install a database, import the files into the database and then use some Haskell library, in order to query the data in the database. You can query and transform the data stored in the CSV files in-place, within your Haskell application; generating new CSV files whenever you wish, by using the DBFunctor package and the powerful Embedded Domain Specific Language – called Julius – that comes with it.

Typical examples of DBFunctor use-cases

- **Build database-less Haskell apps** Build your data processing Haskell apps without the need to import your data in a database for querying functionality or any data transformations. Analyze your CSV files in-place with plain Haskell code (*for Haskellers!*).
- **Data Preparation** I.e., clean-up data, calculate derived fields and variables, group by and aggregate etc., in order to feed some machine learning algorithm (*for Data Scientists*).
- **Data Transformation** in order to transform data from Data Model A to Data Model B (typical use-case *for Data Engineers* who perform ETL*/ELT tasks for feeding Data Warehouses or Data Marts)
- **Data Exploration** Ad hoc data analysis tasks, in order to explore a data set for several purposes such as to find business insights and solve a specific business problem, or to do data profiling in order to evaluate the quality of the data coming from a data source, etc (*for Data Analysts*).
- **Business Intelligence** Build reports, or dashboards in order to share business insights with others and drive decision making process (*for BI power-users*)

ETL stands for *Extract Transform and Load* and is the standard technology for accomplishing data management tasks in Data Warehouses / Data Marts and in general for preparing data for any analytic purposes (Ad hoc queries, data exploration/data analysis, Reporting and Business Intelligence, feeding Machine Learning algorithms, etc.). ELT is a newer variation of ETL and means that the data are first Loaded into their final destination and then the data Transformation runs in-place (as opposed to running at a separate staging area on possibly a different server)).

When to use it?

DBFunctor should be used whenever data manipulation/transformation tasks over tabular data must be performed and we wish to perform them with Haskell code, yielding all the well-known (to Haskellers) benefits from doing that.

DBFunctor provides an in-memory data structure called `RTable`, which implements the concept of a Relational Table (which, simply put, is a set of tuples) and all relevant relational algebra operations (selection, projection, inner join, outer joins, aggregations, group by, set operations etc.).

Moreover, it implements the concepts of *Column Mapping* (for deriving new columns based on existing ones – by splitting, merging, or with any other possible combination using a lambda expression or a function to define the new value) and that of the *ETL Mapping*, which is the equivalent of a "mapping" in an ETL tool (like Informatica, Talend, Oracle Data Integrator, SSIS, Pentaho, etc.). With this powerful construct, one can *build arbitrary complex data pipelines*, which can enable any type of data transformations and all these by writing Haskell code.

What Kinds of Data?

With the term "tabular data" we mean any type of data that can be mapped to an `RTable` (e.g., CSV (any delimiter), DB Table/Query, JSON etc). Essentially, for a Haskell data type `a` to be "tabular", one must implement the following functions:

```
toRTable :: RTableMData -> a -> RTable
fromRTable :: RTableMData -> RTable -> a
```

These two functions implement the "logic" of transforming data type `a` to/from an RTable based on specific RTable Metadata, which specify the column names and data types of the RTable, as well as (optionally) the primary key constraint, and/or alternative unique constraints (i.e., similar information provided with a CREATE TABLE statement in SQL).

By implementing these two functions, data type `a` essentially becomes an instance of the type class `RTabular` and thus can be transformed with the DBFunctor package. Currently, we have implemented a CSV data type (based one the [Cassava](https://github.com/haskell-hvr/cassava) library), in order to enable data transformations over CSV files.

Main Features

- No need to use a database, in order to process your data – you can do it in-place with just your Haskell code. Easy manipulation of your "tabular data" (e.g.,CSV files). Create arbitrary complex data transformation flows for your tabular data with ease, with just common Haskell code.
- Enables Functional Data Management by implementing the Relational Table concept and exposing all relational algebra operations (select, project, (inner/outer)join, set operations, aggregations, group by etc.) , as well as the "ETL Mapping" concept (common to all ETL tools) as Haskell functions and data types.
- Provides an Embedded Domain Specific Language (EDSL) for ETL, called Julius, which enables to express complex data transformation flows (i.e., an arbitrary combination of ETL operations) in a more

friendly manner (a Julius Expression), with Haskell code (no special language for ETL scripting required – just Haskell).
- printf-like formatting function, for printing RTables on screen. Ideal for building command-line applications (e.g., a REPL) for manipulating tabular data (for example see [hCSVDB](https://github.com/nkarag/haskell-CSVDB)).
- It is applicable to any kind of data that can be presented in a tabular form. This is achieved by a simple Type Class interface.

Future Vision

Apart from supporting other "tabular" data types (e.g., database tables/queries) our ultimate goal is, eventually to make DBFunctor the **first declarative library for ETL/ELT**, by exploiting the virtues of functional programming and Haskell strong type system in particular.

Here we use "declarative" in the same sense that SQL is a declarative language for querying data. You only have to state what data you want to be returned and you don't care about how this will be accomplished – the DBMS engine does this for you behind the scenes.

In the same manner, ideally, one should only need to code the desired data transformation from a source schema to a target schema, as well as all the data integrity constraints and business rules that should hold after the transformation and not having to define all the individual steps for implementing the transformation, as it is the common practice today. This will yield tremendous benefits compared to common ETL challenges faced today and change the way we build data transformation flows. Just to name a few:
- ETL code correctness out-of-the-box
- No data quality errors due to ETL developer mistakes
- Self-documented ETL code (your documentation i.e., the source-to-target mapping and the business rules, is also the only code you need to write!)
- Drastically minimize time-to-market for delivering Data Marts and Data Warehouses, or simply implementing Data Analysis tasks.

The above is inspired by the theoretical work on Categorical Databases by David Spivak.

Further reading

https://github.com/nkarag/haskell-DBFunctor

4.12 Data Structures, Data Types, Algorithms

4.12.1 Algebraic graphs

Report by:	Andrey Mokhov
Participants:	Arseniy Alekseyev, Neil Mitchell
Status:	usable, active development

Alga is a library for algebraic manipulation of graphs in Haskell. The underlying theory is presented here.

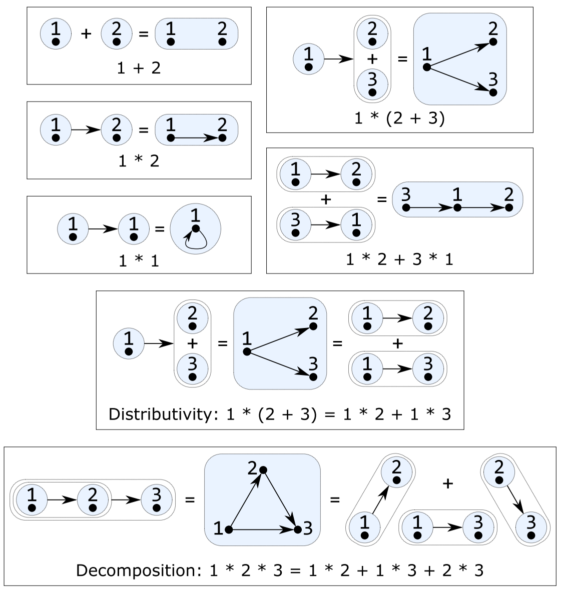
Main idea

Consider the following data type, which is defined in the top-level module Algebra.Graph of the library:

```
data Graph a = Empty
  | Vertex a
  | Overlay (Graph a) (Graph a)
  | Connect (Graph a) (Graph a)
```

We can give the following semantics to the constructors in terms of the pair (V, E) of vertices and edges:
- `Empty` constructs the empty graph $(\emptyset, \emptyset)$.
- `Vertex` x constructs a single vertex, i.e. $(\{x\}, \emptyset)$.
- `Overlay` x y overlays graphs (V_x, E_x) and (V_y, E_y) constructing $(V_x \cup V_y, E_x \cup E_y)$.
- `Connect` x y connects graphs (V_x, E_x) and (V_y, E_y) constructing $(V_x \cup V_y, E_x \cup E_y \cup V_x \times V_y)$.

This figure shows examples of graph construction, where $+$ and $*$ stand for `Overlay` and `Connect`.



1 + 2
1 * (2 + 3)
1 * 2
1 * 1
1 * 2 + 3 * 1
Distributivity: 1 * (2 + 3) = 1 * 2 + 1 * 3
Decomposition: 1 * 2 * 3 = 1 * 2 + 1 * 3 + 2 * 3

We can give an algebraic semantics to the graph construction primitives by defining the type class:

```
class Graph g where
  type Vertex g
  empty :: g
  vertex :: Vertex g → g
  overlay :: g → g → g
  connect :: g → g → g
```

Instances of the type class obey the following laws:

- $(+, \texttt{empty})$ is an idempotent commutative monoid.
- $(*, \texttt{empty})$ is a monoid.
- $*$ distributes over $+$, e.g. $x * (y + z) = x * y + x * z$.
- $*$ can be *decomposed*: $x * y * z = x * y + x * z + y * z$.

This algebraic structure corresponds to unlabelled directed graphs: every expression represents a graph, and every graph can be represented by an expression. The library defines several law-abiding instances and polymorphic graph manipulation functions.

Current status

The library is documented, tested and usable. It is under active development, so the API is subject to change.

Further reading

- http://hackage.haskell.org/package/algebraic-graphs
- https://github.com/snowleopard/alga-paper

4.12.2 JudyGraph

Report by:	Tillmann Vogt
Status:	experimental

Overview

JudyGraph is a graph library that can handle very dense graphs with a million edges per node. As the name suggests, the library is based on Judy arrays, which is a very fast and memory efficient key-value storage. The properties of this key-value storage guided the development in that node and edge data had to be compressed into 32 bit values with typeclasses. The lowest bits of the edge are used to enumerate several edges with the same property (which is encoded in the higher bits). The documentation of Judy Arrays promises very little CPU cache misses when only the lowest bits are used successively. Another trick to squeeze edge data into 32 bit is to interpret an edge differently depending on the node index of the origin node. Depending on the range in which the index is the edge is interpreted differently.

Query EDSL

The library was developed as an alternative for the Neo4j graph database. That's why it also has a query language that resembles Cypher:

```
   query <- temp jgraph
     (person --| raises |-- issue
      --| references |-- issue)
 where
  person = node (nodes32 [0]) :: CyN
  raises = edge (attr Raises) :: CyE
  issue = node (labels [ISSUE]) :: CyN
```

Storing GHC Core of 1000s of libraries

Last year at Hal Leipzig, I showed that storing the ghc-core code of 1000s of libraries in a graph could be of great help for Haskell tooling. The goal is a function search engine that can get better search results by using PageRank. It should also be possible to click together little programs, once a function is picked. The doc folder contains the slides of this talk.

Future

Currently the library is targeted to load a large graph from csv-files (that fits into memory) and to achieve fast queries and do some post processing like adding or updating edges, but no deletion. The future plan is to implement persistence, get ACID guarantees and implement missing Cypher commands. I would be glad for hints.

Further reading

- https://github.com/tkvogt/judy-graph-db
- https://en.wikipedia.org/wiki/Judy_array

4.12.3 Conduit

Report by:	Michael Snoyman
Status:	stable

The conduit package is one of the most popular approaches to solving the streaming data problem in Haskell. It provides a composable, resource-safe, and constant memory solution to many common problems. With the well developed conduit ecosystem around it, you can easily deal with a variety of data sources (files, memory, HTTP, TCP), file formats (XML, YAML, JSON, CSV), advanced abstractions around parallel processing and concurrency, and have access to a wide range of helpful library functions to choose from.

Since the last HCAR, conduit has had a new major release, version 1.3. The changes include:

- Drop monad-control and exceptions in favor of unliftio
- Drop mmorph dependency
- Deprecate old type synonyms and operators (see http://www.snoyman.com/blog/2016/09/proposed-conduit-reskin)
- Drop finalizers from the library entirely
- Add the `Conduit` and `Data.Conduit.Combinators` modules, stolen from `conduit-combinators`

This results in a simpler core, more consistent API, and a more batteries-included experience in the conduit library itself.

Under the surface, conduit makes use of coroutines and an inlined free monad transformer approach to allow for a high level of flexibility in conduit composition. This is as opposed to other approaches, like list-t or stream fusion, which trade in some of that flexibility for performance. Some of these ideas were explored in http://www.yesodweb.com/blog/2016/02/first-class-stream-fusion. The end result is that, for most I/O based applications, conduit provides a great trade-off. For fully CPU-bound operations, you'll likely want to consider using something less flexible but higher performance.

Conduit is intended to be a replacement to usage of lazy I/O in Haskell code, allowing us to work on large data sets, ensure resources are cleaned up promptly, and retain composable programs. Please see the aforementioned tutorial for many examples of how this works.

The conduit package is designed to work well with the resourcet package, which allows for guaranteeing resource finalization in continuation-based monads. This is one of the main simplifications that conduit achieved versus previous streaming approaches, such as the enumerator package and other left-fold iterator approaches.

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 (https://github.com/snoyberg/conduit#readme).

Conduit includes a stream fusion framework for optimizing away intermediate data creation. Unfortunately, these optimizations do not always trigger due to problems with rewrite rule firing. There is some consideration around making this stream fusion a first-class entity that developers can rely upon. See http://www.yesodweb.com/blog/2016/02/first-class-stream-fusion and https://github.com/snoyberg/foreach.

Further reading

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

4.12.4 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.

4.12.5 Concurrent Trie

Report by:	Michael Schröder
Status:	stable

A non-blocking concurrent map implementation based on the *lock-free concurrent hash trie* (aka *Ctrie*).

A hash trie is a tree whose leaves store key-value bindings and whose nodes are implemented as arrays. The concurrent trie extends the hash trie by adding indirection nodes above every array node. Indirection nodes have the property that they stay in the trie even if the nodes above or below them change. When inserting an element into the trie, instead of directly modifying an array node, an updated copy of the array node is created and an atomic compare-and-swap operation on the indirection node is used to switch out the old array node for the new one. If the compare-and-swap operation fails, the insert is retried from the beginning. This simple scheme, where indirection nodes act as barriers for concurrent modification, ensures that there are no lost updates or race conditions of any kind, while keeping all operations completely lock-free.

A more thorough discussion, including proofs of linearizability and lock- freedom, can be found in the papers by Prokopec et al.

Further reading

- http://hackage.haskell.org/package/ctrie
- "Cache-Aware Lock-Free Concurent Hash Tries", Aleksander Prokopec, Phil Bagwell, Martin Odersky
- "Concurrent Tries with Efficient Non-Blocking Snapshots", Aleksander Prokopec, Nathan G. Bronson, Phil Bagwell, Martin Odersky

4.12.6 Random access zipper

Report by:	Li-yao Xia
Status:	Experimental, work in progress

The Random Access Zipper (RAZ) is a data structure for representing sequences with efficient indexing and edits.

The paper introducting it (with an implementation in OCaml) reported performance that is competitive with the more common Finger Tree structure.

I have translated it in Haskell, and started implementing the same interface as `Data.Sequence` from containers in `Data.Raz.Sequence`.

I reproduced the benchmarks from the paper as well as those of `containers`. On average, Haskell's `raz` is slightly slower than OCaml's, but faster than `containers` for many operations.

Future work

The `Data.Raz.Sequence` module remains to be finished to fully match `Data.Sequence`.

There are certainly lots of optimizations opportunities.

Raz requires randomness, which results in some awkward types in Haskell on the one hand (or, internal (ab)use of `unsafePerformIO` to implement a pure interface), but this explicitness can be informative on the other hand, though I am not yet certain how that information may be usefully exploited.

Further reading

- Random Access Zippers: Simple, Purely-Functional Sequences, K. Headley, M. A. Hammer.
  https://arxiv.org/abs/1608.06009
- https://hackage.haskell.org/package/raz
- https://github.com/Lysxia/raz.haskell

4.12.7 Generic random generators

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

Description

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`.

The module `Generic.Random.Generic` leverages `GHC.Generics` to handle common boilerplate in instances of `Arbitrary` for simple datatypes.

However, for recursive datatypes, a naive generator is likely to have problematic issues: non-termination, inconveniently biased distributions (too large, too small, too full). `Generic.Random.Data` derives Boltzmann samplers, 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; the distribution is uniform when conditioned to a fixed size: two values with the same size occur with the same probability. An implementation can now be found in the boltzmann-samplers library (it has been split out of generic-random).

Status

Slowness of Boltzmann samplers I found out that the FEAT library, which can derive random generators for the same class of datatypes producing values of a given size exactly and uniformly distributed, has *much* better performance as well.

A more detailed explanation can be found at https://github.com/Lysxia/generic-random/issues/6.

In theory, Boltzmann samplers have the better asymptotic complexity, but they come with an overhead that appears hard to get rid of; boltzmann-samplers only catches up to FEAT on data sizes that seem too large to be practical (thousands of constructors).

Due to that, I have lost some motivation to go forward with boltzmann-samplers. I still remain open to discussion and suggestions.

Generic and applicative interface In spite of the above issues, I've started a rework of boltzmann-samplers using `GHC.Generics` rather than SYB, and a more general way to obtain generators from an applicative specification.

The branch can currently be found at: https://github.com/Lysxia/boltzmann-samplers/tree/generics.

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
- https://hackage.haskell.org/package/generic-random
- https://github.com/Lysxia/generic-random
- https://hackage.haskell.org/package/testing-feat

4.12.8 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), trees, as well as profile hidden Markov models, and profile stochastic context-free grammars. 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. Currently, ADPfusion, as a core library, provides support for linear string-based grammars. All above-mentioned variants are provided as extension packages and can serve as a guideline on how to expand the capabilities of ADPfusion.

As an example of ADPfusion in practice, 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 $p \to c\,p\,c \mid \epsilon$.

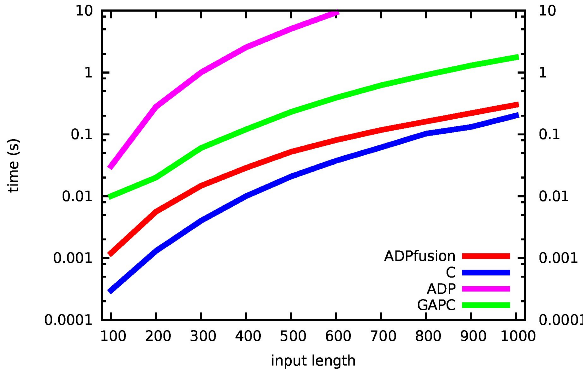
The corresponding ADPfusion code is similar:

```
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\ = $ `True`, and $h\ xs = $ `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 (→ 4.12.9) 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

4.12.9 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, sets, and profile stochastic structures (profile HMMs and profile SCFGs); 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 rigorous 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 (→ 4.12.8) 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.

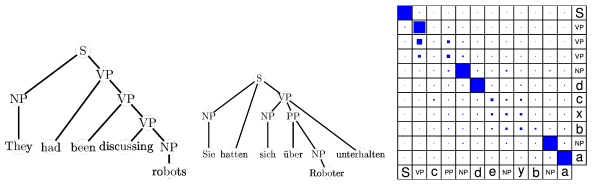
If, say, the user wants to calculate the shortest Hamiltonian path through all nodes of a graph, then the grammar for this problem is:

```
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:


S
VP
NP
They had been discussing robots
Sie hatten sich über Roboter unterhalten
PP

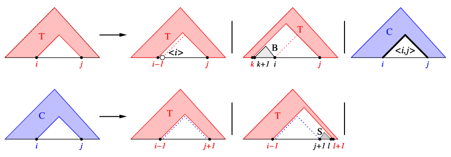
We can create the parse trees themselves with a normal context-free language on sequences. We can also compare the two sentences with, say, a Needleman-Wunsch style sequence alignment algorithm. However, this approach ignores the fact that parse trees encode grammatical structure inherent to languages. 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 (→ 4.12.8) 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/
- Product Grammars for Alignment and Folding https://doi.org/10.1109/TCBB.2014.2326155
- Algebraic Dynamic Programming over General Data Structures https://doi.org/10.1186/1471-2105-16-S19-S2

- Algebraic Dynamic Programming for Multiple Context-Free Languages http://www.sciencedirect.com/science/article/pii/S0304397516301797
- Algebraic Dynamic Programming on Trees https://doi.org/10.3390/a10040135

4.12.10 Applications of Algebraic Dynamic Programming

Report by:	Christian Höner zu Siederdissen
Participants:	Maria Beatriz Walter Costa
Status:	usable, active development

Here, we showcase an application developed using *generalized Algebraic Dynamic Programming* (→ 4.12.9).

Temporal Ordering of Substitutions in RNA Evolution

Joint work with Maria Beatriz Walter Costa, Dan Tulpan, Peter F. Stadler, and Katja Nowick.

The `MutationOrder` program tries to solve the following problem: We are given two RNA input sequences: (i) an ancestral sequence, and (ii) an extant sequence which is related to the ancestral sequence but has undergone a number of mutations (that are typically thought of to be beneficial).

We now have to determine the most likely order in which single nucleotide mutations happened between two RNA sequences. The algorithm that answers this question is a variant of the travelling salesman problem. First, we need to keep track of all the mutations (nodes or "cities") previously visited, instead of just the two nodes indicating the most recent and current change as these modify the cost of the current change. Second, it is biologically possible that one or more hidden nodes – corresponding to mutations and backmutations were visited. This in turns leads to a multi-stage histomorphism problem we solve with our algorithm.

Further reading

- hackage http://hackage.haskell.org/package/MutationOrder
- paper https://www.sciencedirect.com/science/article/pii/S0022519317305222

4.12.11 Earley

Report by:	Olle Fredriksson
Participants:	Spiros Boosalis, Oleg Grenrus, Tero Keinänen
Status:	maintained

Earley is a parsing library that can parse all context-free grammars, including tricky ones for example with left-recursion. The grammars are specified in applicative style.

A new feature in the Earley library is language generation. Given a grammar and a list of allowed input tokens, Earley can generate the members of the language that the grammar generates. The following example shows the language generated by a Roman numerals grammar limited to the tokens 'V', 'I', and 'X'.

```
language (generator romanNumeralsGrammar "VIX")
 = [(0,""),(1,"I"),(5,"V"),(10,"X"),(20,"XX"),
   (11,"XI"),(15,"XV"),(6,"VI"),(9,"IX"),
   (4,"IV"),(2,"II"),(3,"III"),(19,"XIX"),
   (16,"XVI"),(14,"XIV"),(12,"XII"),(7,"VII"),
   (21,"XXI"),(25,"XXV"),(30,"XXX"),
   (31,"XXXI"),(35,"XXXV"),(8,"VIII"),
   (13,"XIII"),(17,"XVII"),(26,"XXVI"),
   (29,"XXIX"),(24,"XXIV"),(22,"XXII"),
   (18,"XVIII"),(36,"XXXVI"),(39,"XXXIX"),
   (34,"XXXIV"),(32,"XXXII"),(23,"XXIII"),
   (27,"XXVII"),(33,"XXXIII"),(28,"XXVIII"),
   (37,"XXXVII"),(38,"XXXVIII")]
```

Further reading

https://github.com/ollef/Earley

4.12.12 Type Providers

Report by:	Michał J. Gajda
Participants:	Michał Gajda, Kevin Cheung, Guru Devanla, and others
Status:	Started

Discussion in the DataHaskell community suggested that we need an extensive effort to make it easier to parse different file formats, and hasten the setup of DataHaskell projects.

Currently we have `json-autotype` (→ 4.9.3), and `Frames` library that autodetect file format, generate appropriate type description, and the parser.

We expanded our efforts to build type providers with `xml-typelift` library written by Kevin Cheung with advice of Michal - it generates type descriptions.

We also started building `type-provider` tool that will catalog the existing parsers for common file formats, and automaticaly generate import code. Prototype is available in GitHub repository.

Further reading

https://github.com/DataHaskell/type-providers

4.12.13 Transient

Report by:	Alberto Gómez Corona
Status:	active development

Transient is a monad/applicative/Alternative with batteries included that brings 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, indeterminism, concurrency, parallelism, thread control and distributed computing, publish/subscribe and client/server side web programming among others.

All effects can be combined while maintaining algebraic and monadic composability using standard applicative, alternative and monadic combinators.

What is new in this report is:

- Restoring execution state from checkpoint
- Nodes can connect using websockets or relay communications
- Secure communications with TLS
- Optimize local calls
- Exception management using backtracking
- HTML rendering support templates and template edition !!??

Future work: Relay communications, Programmer-defined serialization, Server side HTML rendering

Further reading

- gitter chat
- Transient tutorial
- distributed Transient, GIT repository
- 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

4.12.14 Streamly: Streaming Concurrently

Report by:	Harendra Kumar
Status:	Active Development

Streamly is a natural extension of Haskell lists to monadic streaming with concurrent composition:

```
import Streamly
import qualified Streamly.Prelude as S
main = runStream $
      S.repeatM getLine
    & fmap read
    & S.filter even
    & S.takeWhile (<= 9)
    & fmap (\x -> x * x)
    & S.mapM print
```

You can fold streams concurrently.

```
main = S.toList $ parallely $
  foldMap delay [1 .. 10]
  where
    delay n = S.once $
      threadDelay (n * 1000000) >> print n
```

Streams support nested and concurrent looping constructs. For example, to compute each square and the sum of squares concurrently:

```
main = do
  s <- S.sum $ asyncly $ do
    x2 <- foldMap square [1 .. 10]
    y2 <- foldMap square [1 .. 10]
    return $ sqrt (x2 + y2)
  print s
  where square x = return $ x * x
```

The following example recursively lists a directory tree concurrently. To see the elegance of the API, just remove *runStream* and the *asyncly* combinator and the code now becomes regular IO monad code.

```
main = runStream $ asyncly $
  getCurrentDir >>= readdir
  where
    readdir d = do
      (dirs, files) <- liftIO $ listDir d
      liftIO $ mapM_ print files
      foldMap readdir dirs
```

There is much more to Streamly, please see the tutorial included in the package, and the *streaming-benchmarks* (→ 4.12.15) repository on github for performance comparison. There is a lot left to do; contributions are needed and welcome.

Further reading

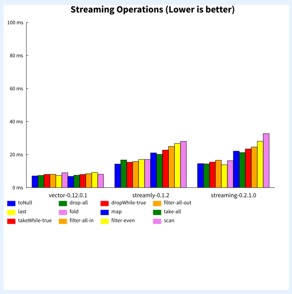
- https://github.com/composewell/streamly
- https://github.com/composewell/streaming-benchmarks

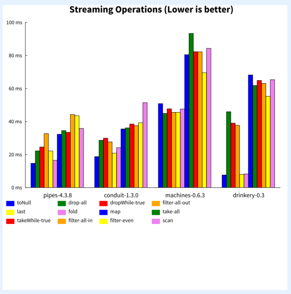
4.12.15 Streaming Performance Benchmarks

Report by:	Harendra Kumar
Status:	Active Development

There are quite a few Haskell streaming libraries providing streaming IO functionality. Apart from the difference in user level APIs they also use different design philosophies and the underlying stream or stream processor types. It is interesting to know how the different approaches compare with each other in terms of performance on micro-benchmarks. In addition to helping users in choosing a library, comparative performance measurement can also provide an insight into the strengths and weaknesses of the design approaches which can in turn help in choosing the best approach when building a new library. It can also help in finding performance issues and improving these libraries.

streaming-benchmarks package provides a common framework to fairly measure and compare the performance of the various monadic streaming libraries including *vector*, *streamly* (→ 4.12.14), *streaming*, *pipes*, *conduit*, *machines* and *drinkery*. The library also provides performance measurement of pure data structures like *list* and *vector* for a comparison with the monadic ones. The benchmarking code is pretty modular and new streaming libraries can be added easily. The figures in this section show some of the benchmarks, please see the README in the *streaming-benchmarks* GitHub repository for more details.





A word of caution for end users, these are micro benchmarks and whether they are significant for a particular use case in a macro setting depends on the contribution of the micro-benchmark cost to the overall performance of the application.

Further reading

https://github.com/composewell/streaming-benchmarks

4.12.16 proto-lens

Report by:	Judah Jacobson
Status:	active

The proto-lens library provides an API for protocol buffers, a language-independent binary file format. Cabal (→ 4.2.1) and Stack (→ 4.2.2) projects can use proto-lens to autogenerate Haskell source bindings from the original protocol buffer specifications.

The library uses modern Haskell language and library patterns for simpler and safer code, including:

- Composable field accessors via lenses and prisms
- Overloaded names for field accessors via type-level literals and the `OverloadedLabels` extension
- Type-safe reflection and encoding/decoding of messages via GADTs

Version 0.3.0.0 of the library was recently released; it includes usability improvements to the API and generating prisms for "oneof" fields (sum types). Ongoing work includes better integration with gRPC and improving the type error messages.

Further reading

- https://hackage.haskell.org/proto-lens
- https://google.github.io/proto-lens
- https://github.com/google/proto-lens

4.13 Parallelism and Concurrency

4.13.1 Eden

Report by:	Jost Berthold and Rita Loogen
Participants:	Rita Loogen, Lukas Schiller (Marburg), Jost Berthold (Sydney), Florian Fey (Münster), Yolanda Ortega-Mallén, Mercedes Hidalgo, Fernando Rubio, Alberto de la Encina (Madrid)
Status:	available

Eden extends Haskell with a small set of syntactic constructs for explicit process specification and creation, including automatic process communication (via head-strict lazy lists) and synchronization. Higher-level coordination is achieved by defining *skeletons*, ranging from a simple parallel map to sophisticated master-worker schemes. Eden's interface includes the concept of *Remote Data*, which provides implicit stream communication channels between processes to compose skeletons and define arbitrary communication topologies. A *PA*-monad enables the *eager* execution of user defined sequences of *Parallel Actions* in Eden.

Standard reference: Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña: *Parallel Functional Programming in Eden*, JFP 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 currently implemented by modifications to the Glasgow-Haskell Compiler GHC, extending its runtime system to use multiple communicating instances with independent heap management. The implementation supports MPI (or PVM) in cluster environments, and a shared memory mode solely based on

OS support (Windows/Linux) on multicore platforms. Source code of the modified GHC is available from http://github.com/jberthold/ghc; there are Eden variants of every major/minor GHC version since 2008, which can be built using only the usual GHC build tools (and MPI if desired).

Building on this runtime support, the Haskell package *edenmodules* defines the language, and *edenskels* provides a library of parallel skeletons. The Eden libraries are available via Hackage and from http://hex.mathematik.uni-marburg.de:8080/.

As an alternative, a library-only implementation of Eden was recently developed by Florian Fey. This implementation uses the *network-fancy* package for networking, and the *packman* package for serialising Haskell heap objects.

If you want to use Eden, we will be happy to provide support, and delighted to receive your contributions to its implementation — please contact us via github or e-mail.

Tools and libraries

The Eden trace viewer tool *EdenTV* provides a visualisation of Eden program runs. Visual post-mortem activity profiles are produced for processing elements (machines), Eden processes and threads, including message traffic. EdenTV is written in Haskell and is freely available on hackage. It can also display thread views of GHC eventlogs.

A second trace viewer tool is *Eden-Tracelab*, by Bastian Reitemeier. It can visualise larger trace files by using an external database. See brtmr.de/2015/10/17/introducing-eden-tracelab.html.

The Eden skeleton library contains various skeletons for parallel processing, including parallel maps, workpools, divide-and-conquer, and common process topologies. Take a look on the Eden pages or the haddock documentation on hackage for an overview.

Applications

Eden's simple API of explicit processes with implicit communication enables rapid prototyping for parallel programs at both the skeleton/library and the application level. A range of research has investigated Eden skeletons library optimisations and refinements. Eden has been used to solve optimisation problems in continuous domains. This has been done by defining parallel skeletons dealing with different bioinspired metaheuristics, such as Particle Swarm Optimization, Differential Evolution, and Artificial Bee Colony.

Further reading

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

4.13.2 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 lightweight, higher-order, polymorphic, pure functional language implementation in which we can experiment with automatic parallelization strategies, varying degrees of default function and constructor strictness, and lightweight instrumentation.

We do not consider speculative or eager evaluation, but do plan to infer strictness by program analysis, so potential parallelism is dictated by the specified degree of default strictness, language constructs for parallelism, and program analysis.

Our approach is similar to that of the Intel Labs Haskell Research Compiler: we use GHC as a front-end to generate STG, then exit to our own back-end compiler; additionally, we have native *Mini-Haskell* and STG front-ends. As in their case we do not attempt to use the GHC runtime. Our implementation is *lightweight* 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. The most recent major development was the bridge between GHC and our system. Thus we can now compile and run Haskell programs with simple primitive and algebraic data types using GHC or our own mini-Haskell front-end.

Additionally, we have developed a new strictness analysis technique that is currently being implemented.

Immediate Plans

We are currently developing a more realistic parallel runtime, and writing up and implementing the new strictness analysis technique.

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, USA, 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. Dates for terms beyond summer 2018 are best guesses based on prior years.

Term	Application Opening	Deadline
Fall 2018	**OPEN NOW**	May 31 2018
Spring 2019	July 2018	Oct 2018
Summer 2019	Oct 2018	Jan 2019

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

Further reading

Email same address as above for the Trends in Functional Programming 2016 paper about this project.
Intern Loren Anderson did an interesting Haskell exercise while here for this mathematics paper.

4.13.3 concurrent-output

Report by:	Joey Hess
Status:	stable, actively developed

A common problem with concurrent programs is that output to the console has to be buffered or otherwise dealt with to avoid multiple threads writing over top of one-another. This is particularly a problem for progress displays, and the output of external processes. The concurrent-output library aims to be a simple solution to this problem.

It includes support for multiple console regions, which different threads can update independently. Rather than the complexity of using a library such as ncurses to lay out the screen, concurrent-output's regions are compositional; it acts as a kind of miniature tiling window manager. This makes it easy to generate progress displays similar to those used by apt or docker.

STM is used extensively in the implementation, which simplified what would have otherwise been a mess of nested locks. This made concurrent-output extensible using STM transactions. See this blog post.

Concurrent-output is used by git-annex, propellor, and xdcc and patches have been developed to make both shake and stack use it.

4.13.4 Déjà Fu: Concurrency Testing

Report by:	Michael Walker
Status:	actively developed

Déjà Fu is a concurrency testing tool for Haskell built on top of the "concurrency" library, which provides a typeclass abstraction over a large subset of the "base" concurrency API, such as:

- Threading
- Capabilities
- Yielding and delaying
- IORefs, MVars, and Software Transactional Memory
- Relaxed memory for IORef operations
- Atomic IORef primitives
- Exceptions

In the last six months, in addition to the usual stream of bugfixes and performance improvements, dejafu has: overhauled the API, throwing out many confusingly similar functions; implemented a snapshotting mechanism, to avoid redundancy in test cases with some nontrivial set-up work; gained support for simplifying execution traces; and switched to using a more convenient record-based approach for configuration.

Further reading

- http://hackage.haskell.org/package/dejafu
- http://hackage.haskell.org/package/concurrency
- 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.

4.14 Systems programming

4.14.1 Haskell for Mobile development

Report by:	Moritz Angermann
Participants:	zw3rk.com and obsidian.systems
Status:	in review

The set of languages to choose from for mobile development is limited to those languages that can target those ecosystems.

There have been ongoing efforts to make Haskell a viable choice for Mobile Development for many years, via the path of cross compilation.

A major obstacle for cross compilation with Haskell is Template Haskell. Up until recently Template Haskell was available only for stage2 compilers, while cross compilers are stage1. One notable exception is GHCJS, which had Template Haskell support quite some time already via an out of process Template Haskell solution. With the addition of `-fexternal-interpreter` via `iserv` in recent GHCs, which implements a very similar system, it is possible to add TH support to stage1 cross compilers.

With the addition of linker for mach-o/aarch64, elf/aarch64, and improvements to the elf/armv7 linker in GHC, as well as a proxy mechanism that allows GHC to communicate with a remote iserv on a different host it is now possible to compile large chunks of TH code with cross compilers targeting supported linker platforms. File and Process IO pose interesting problems.

The necessary changes to GHC are all under review and will hopefully make it into GHC 8.4 in time.

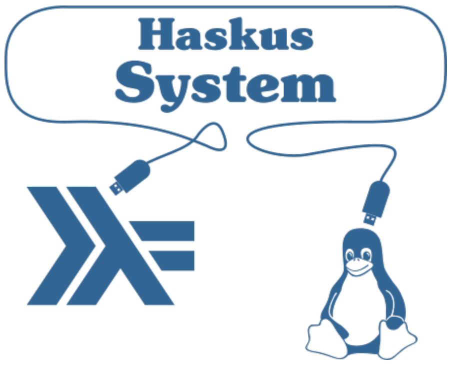
There are some additional challenges in the Haskell ecosystem (e.g. cabal is not very cross compilation aware) that need to be addressed before Haskell for Mobile works out of the box.

A periodically force-pushed snapshot of the GHC HEAD plus the open differentials is available, as well as build scripts that build ghc for iOS and Android using a custom toolchain, derived from the ghc-ios-scripts. GHCSlave apps for iOS and Android that wrap iserv.

Further information covering the ongoing development will be published at https://medium.com/@zw3rk.

4.14.2 haskus-system

Report by:	Sylvain Henry
Status:	active



`haskus-system` is a system programming framework directly on top of the Linux kernel system calls. It doesn't rely on usual interfaces (e.g., libc, libdrm, libinput, X11, wayland, etc.) to communicate with the kernel. Everything is done in Haskell.

Notable changes since the last HCAR:

- ◦ Building a system is much easier. We just have to declare the Linux version to use and some other options in a `system.yaml` file and a tool called `haskus-system-build` automatically downloads, builds and configures the dependencies (Linux, SysLinux).
- ◦ The same tool can also be used to build ISO images and to execute the system with QEMU.
- ◦ The user manual has been enhanced and a new demo is available showing a simple terminal-like application.

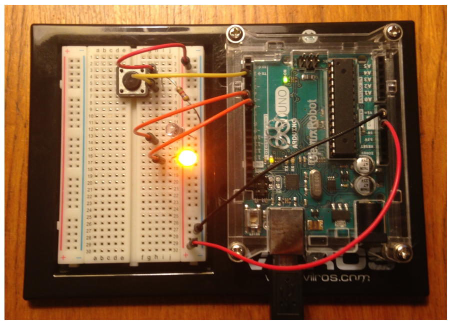
The source code is freely available (BSD3 license).

Further reading

http://www.haskus.org/system

4.14.3 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.

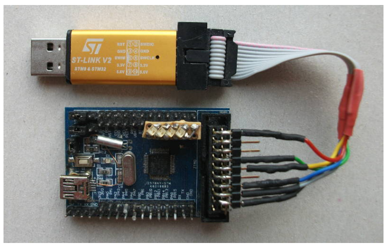
The development has been active over the past year. A paper was published at PADL 2016 for original version, and there is a paper accepted for presentation at TFP 2016 for the new scheduled and compiled version.

Further reading

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

4.14.4 STM32-Zombie

Report by:	Marc Fontaine
Status:	active



The *STM32-Zombie* project turns a `STM32Fxxx` micro controller into a Haskell programmable and flexible IO peripheral of your PC. The `STM32Fxxx` micro controller family features a variety of powerful `IO` peripherals like `GPIO` ports, `USART`, `SPI`, `I2C`, `USB`, `ADC`, timers, real time clock, etc. and *STM32-Zombie* allows a Haskell program to control the complete set of built-in micro controller peripherals.

The project is called *STM32-Zombie* because it shuts down the controllers brain (the `ARM CPU`) and turns it into a remote controlled zombie. It works without any c-code, cross-compiler tool-chain, or firmware. The `STM32Fxxx` peripherals use memory mapped control registers and the on-chip-debugging interface of the controller allows Haskell to access all of the controllers address space and registers. With help of the `DMA` controller even hard-real-time applications, like high-frequency sampling or generating of high-frequency output patterns, are possible.

Minimal hardware requirements, for trying out this project, are a mini `STM32F103` breakout board and a `STLink V2 USB dongle simulator`. Both parts are available for less then $2 each. My test setup are mini `STM32F103` breakout boards. I have not tested the library with other members of the `STM32Fxxx` family but the library is probably a good starting point for work on other `STM32Fxxx` controllers.

The cabal package contains examples for LED blinking, serial port, SPI, high-frequency ADC sampling, control of `WS1228B RGB LED` strips, and more.

The library uses the naming conventions of the *ST-Microelectronics STM32F10x Firmware Library* and provides a mid-level abstraction of the hardware.

While access to all of the peripherals is possible, the abstraction layer is still work-in-progress and does not cover all of the peripherals yet.

Comments, suggestions, experience reports, and patches are welcome. The project is open to any kind of contribution.

Further reading

- https://github.com/MarcFontaine/stm32hs
- http://hackage.haskell.org/package/STM32-Zombie

4.15 Mathematics, Simulations and High Performance Computing

4.15.1 sparse-linear-algebra

Report by:	Marco Zocca
Participants:	
Status:	Actively developed

This library provides common numerical analysis functionality, without requiring any external bindings. It is not optimized for performance yet, but it serves as an experimental platform for scientific computation in a purely functional setting.

Currently it offers :

- iterative linear solvers of the Krylov subspace type, e.g. variants of conjugate gradient such as Conjugate Gradient Squared, BiConjugate Gradient and BiCGSTAB
- linear eigensolvers, based on the QR algorithm and the Rayleigh iteration
- matrix factorizations (namely, LU and QR)
- a number of utility functions such vector and matrix norms, computation of the matrix condition number, Givens' rotation and Householder reflection matrices, and partitioning/stacking/reshaping operations.

The initial motivation for this was on one hand the lack of native Haskell tools for numerical computation, and on the other a curiosity to reimagine scientific computing through a functional lens.

The implementation relies on nested IntMap's from `containers`.

Currently, a new backend based on `accelerate` is under development, along with a generalized interface based on typeclasses, which will allow to decouple algorithms from datastructures. Eventually, the `accelerate`-based backend and interface will be provided as two distinct packages, and `sparse-linear-algebra` will import from both and provide a new reference implementation.

A usage tutorial on the major functionality is available in the README file, and all interface functions are commented throughout the Haddock documentation.

`sparse-linear-algebra` is freely available on Hackage under the terms of a GPL-3 license; development is tracked on GitHub, and all suggestions and contributions are very much welcome.

Further reading

- https://github.com/ocramz/sparse-linear-algebra
- https://hackage.haskell.org/package/sparse-linear-algebra

4.15.2 aivika

Report by:	David Sorokin
Status:	stable

Aivika is a collection of open-source simulation libraries written in Haskell. It is mainly focused on discrete event simulation but has a partial support of system dynamics and agent-based modeling too.

A key idea is that many simulation activities can be modeled based on abstract computations such as monads, streams and arrows. The computations are composing units, which we can construct simulation models from and then run.

Aivika consists of a few packages. The basic package introduces the simulation computations. There are other packages that allow automating simulation experiments. They can save the simulation results in files, plot charts and histograms, collect the statistics summary and so on. There are also packages for distributed parallel simulation and nested simulation based on the generalized version of Aivika.

The core of Aivika is quite stable and well-tested. The libraries work on Linux, OS X and Windows. They are licensed under BSD3 and available on Hackage.

There are plans to find new application fields for the libraries. The core libraries solve a very general task and definitely can be applied to other fields too.

Further reading

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

4.15.3 General Decimal Arithmetic

Report by:	Rob Leslie
Status:	experimental, active development

Haskell's `Float` and `Double` types are often used for floating-point arithmetic, but sometimes they should not be. Although the Haskell Language Report does not require it, these types are typically implemented using a binary floating-point representation, and consequently do not always have an exact correspondence with the way we write a value using decimal notation. For example, the value 0.1 cannot be represented exactly as a binary floating-point value, because there is no integer solution to make $c \times 2^q = 10^{-1}$.

This can lead to subtle errors in computations, as well as the attendant grief and frustration when our programs fail to meet user expectations. It is why Mike Cowlishaw, editor of the IEEE 754 standard for floating-point arithmetic, writes: "While suitable for many purposes, binary floating-point arithmetic should not be used for financial, commercial, and user-centric applications or web services because the decimal data used in these applications cannot be represented exactly using binary floating-point."

Mike Cowlishaw has an entire section of his website devoted to *General Decimal Arithmetic* (URL below), including a complete specification that describes the decimal arithmetic included in the updated IEEE 754-2008 standard. Decimal floating-point arithmetic works just like its binary cousin, except it uses a radix value of 10 internally instead of 2. This allows decimal floating-point values to be represented exactly; there is a one-to-one correspondence between the way a value is written in decimal notation and the way it is represented internally. Consequently there are fewer surprises when performing arithmetic with such values.

The *decimal-arithmetic* package is being developed to provide a Haskell implementation of the *General Decimal Arithmetic* specification. It offers a versatile `Decimal` type constructor that is parameterized with both a precision and a rounding algorithm; all arithmetic using this type will be restricted to the given precision, using the given rounding algorithm whenever the result of a computation exceeds the precision.

Several type aliases are provided as a convenience. For example, `Decimal64` is a decimal floating-point type with 16 digits of precision (comparable to `Double`) that rounds half even. It is special because it also has a `Binary` instance with a 64-bit encoded representation using the *decimal64* interchange format. Similar types `Decimal32` (cf. `Float`) and `Decimal128` are also provided.

All the usual numeric, fractional, real, and floating-point type classes have been fully implemented, allowing decimal types to be used as drop-in substitutes for `Float` and `Double`. In addition, a low-level arithmetic monad is provided within which primitive operations can be performed with complete control over the outcome of exceptional conditions such as inexact results or division by zero.

The package is currently in a usable state; extensive testing is encouraged. While the package includes a moderate test suite, future work will focus on ensuring the implementation passes all of the test cases provided on Mike Cowlishaw's website.

Further reading

- http://speleotrove.com/decimal/
- https://hackage.haskell.org/package/decimal-arithmetic

4.16 Graphical User Interfaces

4.16.1 wxHaskell

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

wxHaskell

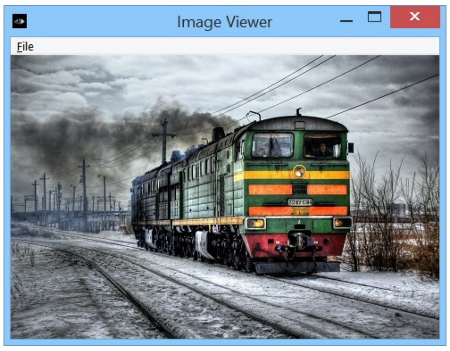
wxHaskell 0.93.0.0 development is in progress, with, amongst others, an adaptation to Cabal 2.0. A cabal.project file is added to the GitHub repository and the test sets are converted to Cabal packages, to be able to compile everything with the experimental command "cabal new-build all". Preparations have been made for the wxWidgets 3.1 binding, but more work is needed for this.

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

4.16.2 threepenny-gui

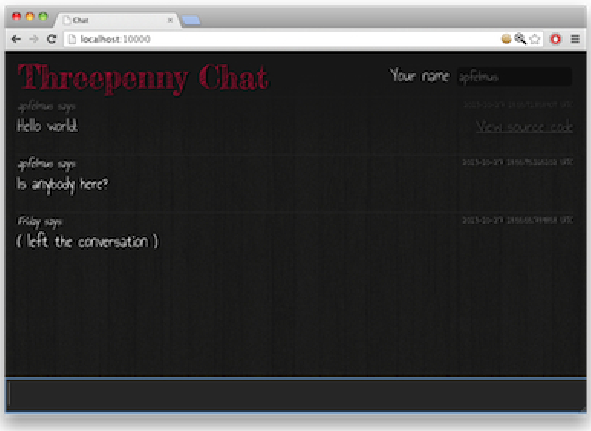
Report by:	Heinrich Apfelmus
Status:	active development

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.

You can download the library from Hackage or Stackage and use it right away to write that cheap GUI you need for your project. Here a screenshot from the example code:



apfelmus says:
Hello world

apfelmus says:
Is anybody here?

Friday says:
(left the conversation)

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

Status

The latest release is version `0.8.2.0`. I would like to thank co-maintainer Simon Jakobi for his diligent and timely releases.

Compared to the previous report, several small additions to the API have been made, the documentation has been improved, and compatibility with the latest GHC versions has been ensured. Moreover, I given a tutorial about the library on HaL 2017, and the slides are now available in the source repository.

Future development

The library is still in flux, API changes are likely in future versions.

In the future, I hope to improve the functional reactive programming (FRP) aspects of the framework, for instance by using the `reactive-banana` library as a basis. (→ 4.17.2)

Further reading

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

4.17 FRP

4.17.1 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).

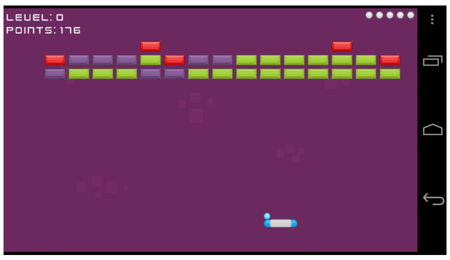
Yampa systems are defined as combinations of Signal Functions. Yampa includes combinators to create constant signals, apply pointwise (or time-wise) transformations, access the running time, introduce delays and create loopbacks (carrying present output as future input). Systems can be dynamic: their structure can be changed using *switching* combinators, which apply a different signal function at some point in the future. Combinators that deal with collections enable adding, removing, altering, pausing and unpausing signal functions at will.

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 a component's internal state. Components can, therefore, be serialized, applied in parallel, etc. Yampa's signal functions implement the Arrow and ArrowLoop typeclasses, making it possible to use both arrow notation and arrow combinators.

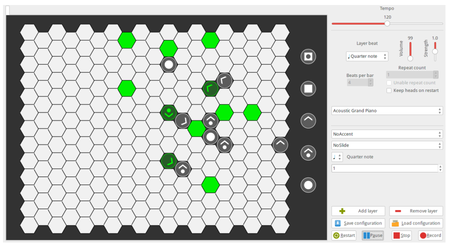
Yampa combinators guarantee *causality*: the value of an output signal at a time t can only depend on values of input signals at times $[0, t]$. Efficiency is provided by limiting history only to the immediate past, and letting signals functions explicitly carry *state* for the future. Unlike other implementations of FRP, Yampa enforces a strict separation of effects and pure transformations: all IO code must exist outside Signal Functions, making 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, which works on Windows, Linux, Mac, Android, iOS and web browsers (thanks to GHCJS). Examples of the latter include Keera Studios (→ 5.2)' Magic Cookies!, a Haskell puzzle board game for iOS and Android available on iTunes (https://goo.gl/6gB6sb) and Google Play (https://goo.gl/0A8z6i).



Guerric Chupin (ENSTA ParisTech), under the supervision of Henrik Nilsson (Functional Programming Lab, University of Nottingham (→ 6.6)) has developed Arpeggigon (→ 4.18.4) (https://gitlab.com/chupin/arpeggigon), an interactive cellular automaton for composing groove-based music. The aim was to evaluate two reactive but complementary frameworks for implementing interactive time-aware applications. Arpeggigon uses Yampa for music generation, Gtk2HS for Graphical User Interface, jack for handling MIDI I/O, and Keera Hails to implement a declarative MVC architecture, based on *Reactive Values and Relations* (RVRs). The results have been written up in an application paper, *Funky Grooves: Declarative Programming of Full-Fledged Musical Applications*, presented at PADL 2017. The code and an extended version of the paper are publicly available (https://gitlab.com/chupin/arpeggigon). Arpeggigon has also been demonstrated at FARM 2017, the Haskell eXchange 2017, and Haskell in Leipzig 2017.



Yampa is under active development, with many Haskellers participating and sending their contributions. Recent releases have featured a cleaner API and new Signal Function combinators. The most recent version, released this year, includes full documentation. Our github repository includes development branches with features that have been used to extend Yampa for custom games. Some of these features have been presented in the paper "Back to the Future: time travel in FRP", presented at the Haskell Symposium 2017, and will be included in future versions. Yampa will now be extended with testing and debugging features, which allow us to express temporal assertions about FRP systems using Temporal Logic, and to use QuickCheck to test those properties, as described in the paper "Testing and Debugging Functional Reactive Programming", presented at ICFP 2017.

Extensions to Arrowized Functional Reactive Programming are an active research topic. Last year we published, together with Manuel Bärenz, a monadic arrowized reactive framework called Dunai (https://git.io/vXsw1), and a minimal FRP implementation called BearRiver. BearRiver provides all the core features of Yampa, as well as additional extensions. We have demonstrated the usefulness of our approach and the compatibility with existing Yampa games by using BearRiver to compile and execute the Haskanoid and Magic Cookies! for Android without changing the code of such games. These games are also available for iOS and other platforms.

The Functional Programming Laboratory at the University of Nottingham (→ 6.6) is working on other extensions to make Yampa more general and modular, increase performance, enable new use cases and address existing limitations. To collaborate with our research, please contact Ivan Perez () and Henrik Nilsson ().

There are several other channels that anyone can use to reach other Yampa users and implementors, including a mailing list and the `#yampa` IRC channel on freenode. We are also active on Haskell Café and the facebook group Programming Haskell, and subscribe to the Yampa keyword on StackOverflow. We encourage all Haskellers to participate on Yampa's development 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. We thank the kind users who have already sent us their contributions.

4.17.2 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

The latest release is version `1.1.0.1`. Thanks to the efforts of our co-maintainer Oliver Charles, the library is available on Stackage. (→ 4.2.3) The library API is stable and feature complete.

Future development

There still remains some work to be done to improve the constant factor performance of the library. Also, the library does not yet compile well to JavaScript with GHC/JS, as there are some issues with garbage collection.

Several other improvements have been suggested on the issue tracker, most notably renaming the `Event` type to `Events`, as the plural provides a more apt description of the semantics of this type.

Further reading

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

4.17.3 Functional Reactive Agent-Based Simulation

Report by:	Jonathan Thaler
Status:	Experimental, active development

Agent-Based Simulation (ABS) is a methodology in which a system is simulated in a bottom-up approach by modelling the micro interactions of its constituting parts, called agents, out of which the global macro system behaviour emerges. So far mainly object-oriented techniques and languages have been used in ABS. We investigate how ABS can be implemented in a pure functional language like Haskell. We build on the concept of Functional Reactive Programming and Monadic Stream Functions for which we use the library Dunai.

It is of most importance to us to keep our code pure - except from the reactive main-loop all our code is pure and does not make use of the IO Monad. This guarantees us reproducibility of the simulation already at compile time because no external sources of non-determinism can influence the computations. We claim that, in contrast to traditional object-oriented languages, this representation is conceptually cleaner and opens the way to formally reason about ABS. For an introduction into the concepts behind functional reactive ABS we refer to our paper, submitted for Haskell Symposium 2018: https://github.com/thalerjonathan/phd/blob/master/public/purefunctionalepidemics/pfe.pdf.

We implemented the library *chimera* for this purpose which implements all concepts (and more) described in the paper. As use-cases we also implemented a number of various well known ABS models e.g. Schelling Segregation, Sugarscape, Agent_Zero. The code is freely available but not stable as it currently serves for prototyping for gaining insights into the problems faced when implementing ABS in Haskell. The plan is to release the final implementation at the end of the PhD as a stable and full-featured library on Hackage.

If you are interested in the on-going research please contact Jonathan Thaler (jonathan.thaler@nottingham.ac.uk).

Further reading

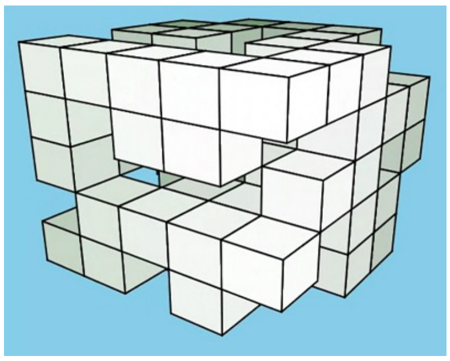
Repository: https://github.com/thalerjonathan/chimera

4.18 Graphics and Audio

4.18.1 diagrams

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

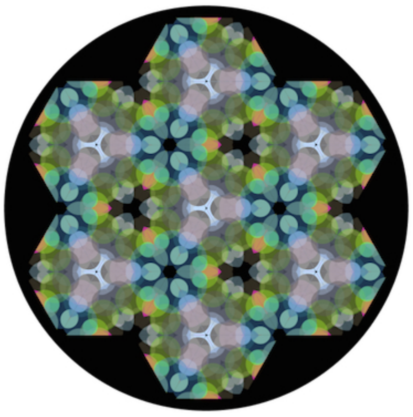
The diagrams framework provides an embedded domain-specific language for declarative drawing. The overall vision is for diagrams to become a viable alternative to DSLs like MetaPost or Asymptote, but with the advantages of being *declarative*—describing what to draw, not how to draw it—and *embedded*—putting 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

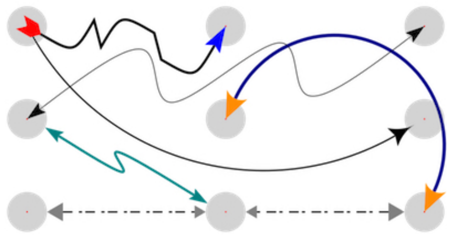
Work on diagrams has slowed considerably since the release of diagrams 1.4 in October 2016, due to time constraints of the main developers. However, work on diagrams 2.0 is slowly but steadily progressing, targeting a release during the summer of 2018. Updates will include:

- Completely rewritten support for animations, with much better semantics and updated examples and tutorials.
- Death to the type-level "backend token", which will allow much easier creation of diagrams that simultaneously work with multiple backends.
- A complete rewrite of the library internals, resulting in better performance and enabling cool new features like diagram traversals.
- Lots of small updates and improvements.



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

4.18.2 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. With CE we can generate the Csound code out of high-level functional description. The project is available on Hackage and can be installed with cabal.

The new version 5.3 is out. Let's look at the new features.

The project was updated to compile with new GHC 8.4.x. Also it was tested on previous compilers down to 7.8. So the CE should compile across 7.8 to 8.4 GHC compilers.

The library was updated to support latest Csound stable release 6.10. There are many new DSP algorithms available with this update. Among them there are many great filters like emulation of Korg 35 analog filter, or emulation of Roland TB-303 resonant filter, zero-delay feedback filters. You can find them at the module Csound.Air.Filter.

This release features new effects useful for guitars. Like emulation of Roland Space echo (function tapeEcho or magnus) and ambient guitar effect (ambiEnv, ambiGuitar). The Space echo simulates behaviour of magnetic tape delay. Ambient guitar detects strike attacks in the audio signal and smoothes them down, so that they sound like pads.

New addition is built in BPM synchronization. User can set global BPM with function setBpm. Then it's possible to use functions that synchronize Hzs (function syn) and seconds (function takt) to global BPM. It's useful to align delay times and LFO rates with global BPM. Also module csound-sampler was updated to respond to changes in global BPM.

There are some additions to improve usability of the library like adding new instances for rendering to Csound files. Like rendering functions with arbitrary number of inputs and outputs and rendering of functions augmented with UIs.

Also documentation, examples and tutorial were updated for recent changes.

New useful functions: brown for brownian noise, resizeGui for scaling GUIs window size.

You can listen to the music that was made with Haskell and the library csound-expression:

- https://soundcloud.com/anton-kho
- https://soundcloud.com/kailash-project

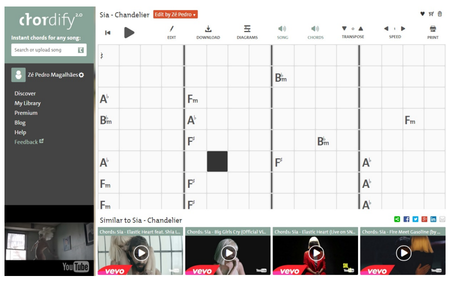
The library is available on Github and Hackage. See the packages csound-expression, csound-sampler and csound-catalog.

Further reading

- https://github.com/spell-music/csound-expression
- http://hackage.haskell.org/package/csound-expression
- http://csound.github.io/

4.18.3 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 have a chord extraction pipeline that is written in Haskell with some bindings to C libraries. We use Kiss FFT for computing audio features, which give information about the frequencies in the audio signal. These features are the inputs for a deep convolutional neural network that is trained and evaluated with bindings to the Tensorflow library. Finally a Hidden Markov Model, which is also trained on datasets of audio with manually annotated chords, is used to pick the final chord for each beat. This model encapsulates the rules of tonal harmony.

We have a distributed backend based on Cloud Haskell, allowing us to easily scale up when the demand increases. Our library currently contains about 6.5 million songs, and about 7,500 new songs are *Chordified* every day. We have also released an iOS app that allows iPhone and iPad users to interface with our technology more easily, and an Android version is being developed.

Chordify is a proud user of Haskell, but we have also encountered some problems and limitations of the language and the libraries. These include:

- A hard-to-find memory leak, where the memory usage of one of our live systems grew slowly over time. After many failed debugging and profiling attempts, this turned out to be a library that was a bit too lazy in evaluating its data. Using a different library with slightly stricter evaluation solved this problem.

- The signal processing libraries that we tried are not efficient and complete enough. At Chordify we want to do fast audio processing, for which Haskell implementations are not available or nowhere near the performance of C libraries.

The code for our old backend, called HarmTrace, is available on Hackage, and we have ICFP'11, ISMIR'12, ISMIR'16 and DLM'17 publications describing some of the technology behind Chordify.

Further reading

https://chordify.net

4.18.4 The Arpeggigon

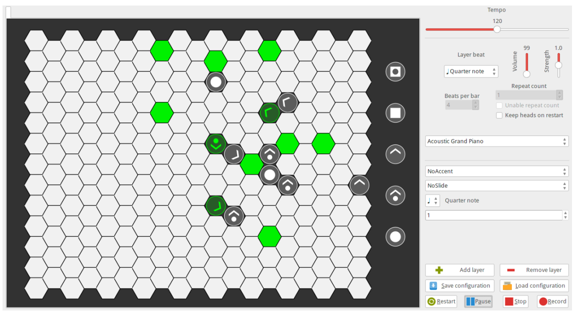
Report by:	Henrik Nilsson
Participants:	Henrik Nilsson, Guerric Chupin, Jin Zhan
Status:	Experimental: working but not yet feature complete

The Arpeggigon is a Functional Reactive Musical Automaton developed by Henrik Nilsson and summer interns Guerric Chupin and Jin Zhan. The work took place in the Functional Programming Laboratory, University of Nottingham (→ 6.6), which Guerric now has joined as a PhD student.

The Arpeggigon was developed as a case study for evaluating the combination of two reactive but complementary frameworks for implementing interactive time-aware applications: (FRP) as embodied by Yampa and Reactive Values and Relations (RVR). Hybrid (mixed continuous and discrete time) aspects were of particular interest.

We choose a musical application as music is a domain with hybrid temporal aspects at its core. The goal was to develop a full-fledged application in terms of look and feel and interoperability with standard MIDI software and hardware as would be found in a studio setting. To that end, we opted for using the standard GUI framework GTK+ for the user interface and the Jack framework for MIDI connectivity. Additionally, we wanted an application that is interesting and fun in its own right.

The Arpeggigon was inspired by the hardware reacTogon: a "chain reactive performance arpeggiator". It's essentially a cellular automaton based around the Harmonic Table: a way to arrange musical notes on a hexagonal grid. *Play heads* bounce between *tokens* arranged on the hexagonal grid. Whenever a play head hits a token, a note corresponding to the position of the token on the grid is played. The picture below shows what the Arpeggigon looks like:


Tempo 120
Layer beat: Quarter note
Volume 99
Strength 1.0
Beats per bar: 4
Repeat count: 1
Unable repeat count
Keep heads on restart
Acoustic Grand Piano
NoAccent
NoSlide
Quarter note
1

The case study demonstrates that FRP in combination with RVR is a compelling combination for building this kind of applications. FRP, in the synchronous dataflow tradition, aligns with the temporal and declarative nature of music, while RVR allows declarative interfacing with external components as needed for full-fledged musical applications.

The results have been written up in an application paper, "Funky Grooves: Declarative Programming of Full-Fledged Musical Applications", that was presented at PADL 2017. There is also an extended technical report version and videos as well as slides from talks (PADL, FARM, Haskell eXchange, etc).

The code is publicly available on GitLab.

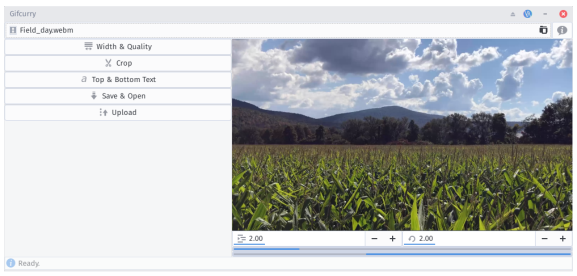
The next steps are to make the Arpeggigon more feature complete and to improve its interoperability in the context of a MIDI studio by improving its timing.

Further reading

- Paper:
  http://eprints.nottingham.ac.uk/38747/
- Technical Report:
  http://eprints.nottingham.ac.uk/38657/
- Haskell eXchange 2017 talk (video)
- London Haskell Meetup talk (slides):
  http://www.cs.nott.ac.uk/~psznhn/talks.html#lhmjune2017
- Computerphile overview:
  https://www.youtube.com/watch?v=v0HIkFR1EN4
- Video Demo:
  https://www.youtube.com/watch?v=yJteVN8OQYk
- Repository:
  https://gitlab.com/chupin/arpeggigon

4.18.5 Gifcurry

Report by:	David Lettier
Status:	actively maintained



Gifcurry is an open source video to GIF maker written in Haskell. Powered by the core library, Gifcurry has both a command-line and graphical user interface. Features include a video preview, seeking, trimming, croping, text overlays with font selection, setting the width and output quality, and saving the results as a GIF or video. Platforms supported include Linux and macOS.

Gifcurry is actively maintained and continuously improved. A two year old project, Gifcurry has had 17 releases at the time of this writing. The most recent version was released in April of 2018.

Gifcurry's source is on GitHub and Hackage. Linux users can download and install Gifcurry via the Arch User Repository, Snapcraft, AppImageHub, or the GitHub releases page. There is a convenient build script for macOS users and a Homebrew package is planned.

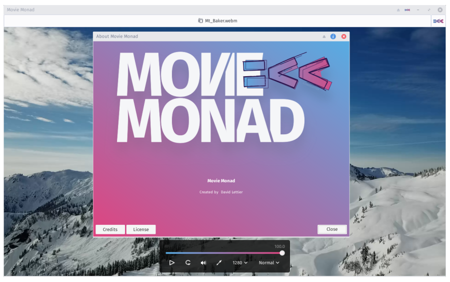
Future improvements include being able to add multiple text overlays and having them show in the video preview. Currently, Gifcurry allows you to add two text overlays (one for the top and bottom) that last the entire duration. Instead, you'll be able to add as many as you wish, specifying the start time, duration, font, and location for each.

Further reading

https://github.com/lettier/gifcurry

4.18.6 Movie Monad

Report by:	David Lettier
Status:	actively maintained



Movie Monad is an open source desktop video player written in Haskell. Its run-time dependencies include GTK+ and GStreamer. To interface with GTK+ and GStreamer, Movie Monad uses the haskell-gi bindings. Features include the usual playback and volume controls, full screen mode, subtitle support, variable speed playback, and a command line interface. Users can play both local and remote files from the web.

While there already exists a wide array of video players, there is still room for Movie Monad with its intuitive interface and unobtrusive feature set. Movie Monad falls somewhere between mpv's minimalism and VLC's verbosity. The main reason for the project's existence, however, is to showcase Haskell's capabilities of targeting the desktop by using Haskell to build a nontrivial, multimedia GUI application.

The project is still ongoing and actively maintained. Originally, Movie Monad was web based. It used Fay, Clay, and Blaze to generate the JavaScript, CSS, and HTML needed to run the program with Electron. However, the project switched to GTK+ and GStreamer for portability and performance reasons.

Movie Monad's source is on GitHub and Hackage. Linux users can download and install Movie Monad via the Arch User Repository, Flathub, Snapcraft, AppImageHub, GNOME Software, or the GitHub releases page. There is a convenient build script for macOS users and a Homebrew package is planned.

Two new features are planned for Movie Monad. In addition to browsing through local files, Movie Monad will support browsing through podcast feeds. The other planned feature involves image recognition allowing users to search through a video much like they would a text document. If you see a feature Movie Monad should have, be sure to leave your feedback on the GitHub issues page.

Further reading

https://github.com/lettier/movie-monad

4.19 Games

4.19.1 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. The players can submit new rules or modify existing ones, thus completely changing the behavior 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.

The V1.0 has been released!

- ◦ a completely new web GUI
- ◦ a library of ready-made rules
- ◦ add client command line interface
- ◦ a REST API
- ◦ a new event programming library called 'Imprevu'
- ◦ container-based deployment

The game has been presented in Zurihac 2017, where we playing several live matches. A lot of learning material is available, including videos, a tutorial, a FAQ, and API documentation.

If you like Nomyx, you can help! There is a development mailing list (check the website).

Further reading

http://www.nomyx.net

4.19.2 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.

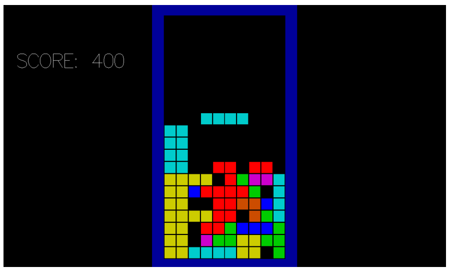
Latest development of EtaMOO can be seen on GitHub, with periodic releases also being made available through Hackage.

Further reading

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

4.19.3 Tetris in Haskell in a Weekend

Report by:	Michael Georgoulopoulos
Status:	actively developed



I made a Tetris in Haskell, while learning the basics of the language, in order to gain some hands-on experience, and also to convince myself that Haskell is a practical language that's worth the time investment and the steep learning curve.

I am now convinced that that is the case. In fact, I'm amazed at how concise and readable Haskell code can be, and I can already acknowledge Haskell as a tool for productivity, predictability and reliability, which are without a doubt, properties most software developers could benefit from.

I have documented this experience as a series of thoughts from the point of view of a beginner, in the form of a blog post titled "Tetris in Haskell in a Weekend"

I also documented the project's evolution in small increments as a git repository that might be of interest to other beginners. The repository can be accessed via github, and contributions are welcome

Further reading

https://github.com/mgeorgoulopoulos/TetrisHaskellWeekend

4.19.4 Barbarossa

Report by:	Nicu Ionita
Status:	actively developed

Barbarossa is a UCI chess engine written completely in Haskell. UCI is one of the two most used 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 v0.4.0 from December 2016.

Further reading

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

4.19.5 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 idioms 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)

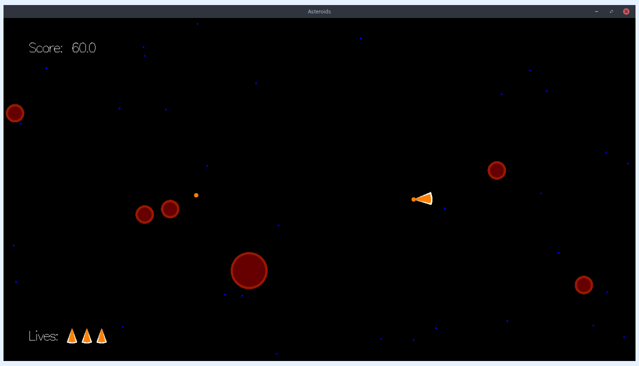
4.19.6 Asteroids

Report by:	Kareem Salah
Status:	available, actively developed

Asteroids is a Haskell version of the known game with the same name. It was developed as learning project.

The main aim of the game is to score the highest score by shooting the coming asteroids while avoiding them and remembering that the player only has three spaceships.

The implementation simulates the space physics and follows the original rules. In addition, we added a multi-player system to invite friends to join the campaign.



Our team found it a good opportunity to develop this using Haskell. By using lazy evaluation techniques we were able to provide a stable framerate.

Further reading

https://github.com/kareem2048/Asteroids

4.20 Data Tracking

4.20.1 hledger

Report by:	Simon Michael
Status:	stable, actively developed

`hledger` is a set of cross-platform tools (and Haskell libraries) for tracking money, time, or any other commodity, using double-entry accounting and a simple future-proof text file format.

It is an enhanced, well-documented reimplementation of plain text accounting in Haskell, inspired by John Wiegley's Ledger program.

`hledger` aims to be a reliable and practical tool for daily use, and provides command-line, curses-style, and web interfaces.

`hledger` is released under GNU GPLv3+.

Project activity has been continuous since our last update in November 2016.

Some notable developments:

- The community has grown in size and activity. The number of committers has doubled, from 42 to 84.
- Website, documentation, CI and project infrastructure have been improved
- A cross-platform installer script has been added
- Many fixes, refinements, and new features have been added, such as:
- balance assignments
- pivot
- periodic transactions
- automated postings
- budget reports
- normal-positive reports
- HTML output
- custom account sorting
- import command
- live reloading in hledger-ui

hledger is available from the hledger.org website, Github, Hackage, or Stackage. It is packaged for a number of systems (Windows, Homebrew, most GNU/Linux distros, NixOS, Sandstorm..) and buildable on other systems supporting GHC (freeBSD, openBSD..).

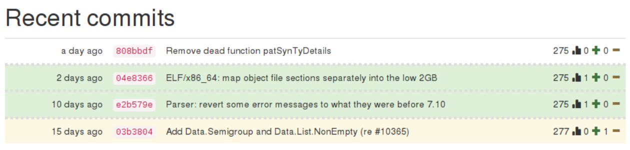
Further reading

http://hledger.org

4.20.2 gipeda

Report by:	Joachim Breitner
Status:	active development

Gipeda is 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.


Recent commits

a day ago	808bbdf	Remove dead function patSynTyDetails	275	0	0	
2 days ago	04e8366	ELF/x86_64: map object file sections separately into the low 2GB	275	1	0	
10 days ago	e2b579e	Parser: revert some error messages to what they were before 7.10	275	1	0	
15 days ago	03b3804	Add Data.Semigroup and Data.List.NonEmpty (re #10365)	277	0	1	

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.

Further reading

https://github.com/nomeata/gipeda

4.20.3 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 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/

4.20.4 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 DebianLike
```

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.

Propellor recently has been extended to support FreeBSD, and this led to Propellor properties including information about the supported OSes in their types. That was implemented using singletons to represent the OS, and functions over type level lists. For details, see this blog post.

Propellor has also been extended to be able to create bootable disk images. This allows it to not only configure existing Linux systems, but manage their entire installation process.

Further reading

http://propellor.branchable.com/

4.21 Others

4.21.1 leapseconds-announced

Report by:	Björn Buckwalter
Status:	stable, maintained

The leapseconds-announced library provides an easy to use static LeapSecondMap 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 2017.1 of leapseconds-announced was released to support the change from LeapSecondTable to LeapSecondMap in time-1.7. It contains all leap seconds up to 2017-01-01. A new version will be uploaded if/when the IERS announces a new leap second.

Further reading

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

4.21.2 clr-haskell (Haskell interoperability with the Common Language Runtime)

Report by:	Tim Matthews
Participants:	José Iborra López
Status:	experimental, actively developed

clr-haskell is a project to enable the use of code within the common language runtime (.NET / Mono / CoreCLR) from GHC Haskell.

This project provides 2 primary flavours for a developer to interop between the CLR & Haskell:

The Haskeller's strongly typed flavour. Takes advantage of the latest GHC extensions to provide a way of encoding an OO type system within the Haskell type system.

The .NET dev's inline flavour. Provides the ability to call directly into valid C# / F# syntax via quasiquoted template Haskell.

Further reading

https://gitlab.com/tim-m89/clr-haskell

4.21.3 Kitchen Snitch server

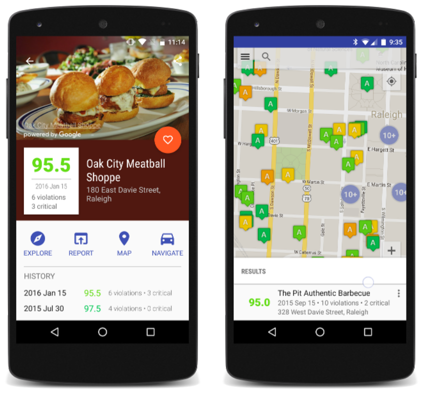
Report by:	Dino Morelli
Participants:	Betty Diegel
Status:	stable, actively developed

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

4.21.4 FRTrader

Report by:	Dimitri DeFigueiredo
Status:	active

FRTrader is a functional reactive bitcoin trading bot. It uses the reactive-banana FRP library and currently has bindings to the GDAX bitcoin exchange. The bot uses as a multi-threaded architecture that makes it easy to plug in extra exchanges and to trade on multiple exchanges simultaneously.

The code is available on github.

The following talk recorded at BayHac 2017 explains the design rationale and the use of FRP in trading.

We welcome contributions and hope to add bindings to other exchanges soon. Also, feel free to make a huge trading profit! Enjoy!

4.21.5 Hapoid

Report by:	Wisnu Adi Nurcahyo
Status:	Under development

Hapoid is an Portable Object Translation file Linter for Bahasa Indonesia.

Currently, this project is a prototype but it will continue to be developed.

Further reading

https://github.com/wisn/hapoid

4.21.6 Hanum - OSM Dynamic Attributes Linter

Report by:	Wisnu Adi Nurcahyo
Status:	Prototype

Hanum is an OpenStreetMap dynamic attributes linter with custom presets.

Some contributors to OSM may just want to fix wrong attributes on the OSM data. This means that they might not want to see any path or shape of the data. This linter thus acts as a library which allows data validation and creating new editors for OSM.

The linter filters error, thus reducing possible conflicts when submitting data to OSM. To further enhance the capabilities, custom presets and rules can be created for each individual country.

Motivation

In Indonesia, especially in Kalimantan there are many areas where OSM has invalid data. For example, SMAN 1 Bintang Ara, which is a senior high school, shows on OSM as having a $school : type\_idn$ attribute of *elementary school*. Furthermore, it also lacks an address attribute.

A similar example is found in Papua: there are locations where the $admin\_level$ attribute is not a number as it should be, as well as areas where $addr : full$ has invalid formatting.

With Hanum, we can define custom rules and use these presets to improve data quality.

Further reading

https://github.com/wisn/hanum

4.21.7 shell-conduit

Report by:	Sibi Prabakaran
Participants:	Chris Done
Status:	active

shell-conduit allows writing shell scripts with conduit. It uses template Haskell to bring all the executables in the PATH as top level functions which can be used to launch them as a process.

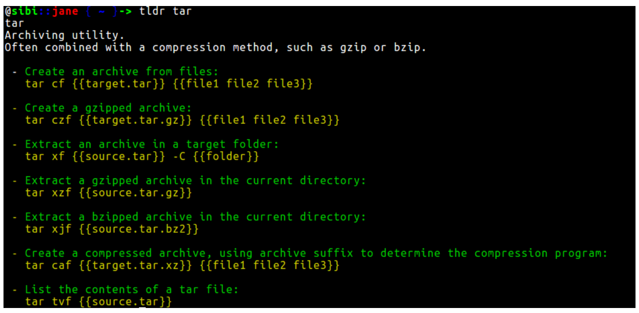
Further reading

- https://github.com/psibi/shell-conduit
- http://chrisdone.com/posts/shell-conduit

4.21.8 tldr

Report by:	Sibi Prabakaran
Status:	active

tldr is a command line client for the TLDR pages. The TLDR pages are a community effort to simplify the beloved man pages with practical examples.



```
@sibi::jane { ~ }-> tldr tar
tar
Archiving utility.
Often combined with a compression method, such as gzip or bzip.

 - Create an archive from files:
   tar cf {{target.tar}} {{file1 file2 file3}}

 - Create a gzipped archive:
   tar czf {{target.tar.gz}} {{file1 file2 file3}}

 - Extract an archive in a target folder:
   tar xf {{source.tar}} -C {{folder}}

 - Extract a gzipped archive in the current directory:
   tar xzf {{source.tar.gz}}

 - Extract a bzipped archive in the current directory:
   tar xjf {{source.tar.bz2}}

 - Create a compressed archive, using archive suffix to determine the compression program:
   tar caf {{target.tar.xz}} {{file1 file2 file3}}

 - List the contents of a tar file:
   tar tvf {{source.tar}}
```

Compared to the previous version, the new verion works in Windows and doesn't do eager cloning. Also default completion support has been added from optparse-applicative. The new-format branch contains changes for the new syntax of TLDR pages and will be merged with master whenever the upstream changes are finalized and merged.

Further reading

https://github.com/psibi/tldr-hs

4.21.9 pprjam

Report by:	Ben Sima
Status:	Beta

`pprjam` is a reference manager for academics that allows you to organize your library of papers, add tags and notes, and create "ppr stacks" (like a playlist for references) to share with your colleagues. More collaboration features are planned for business (paid) users. The feature I'm really excited about, however, is what

I call the “ppr trail” which uses a graph database to show, for any given paper, all of it’s citations and everything that cites it; this would allow you to explore the history of an idea by following the trail of publications over time with just a few clicks.

The first beta prototype has been deployed and is in active development. Production (i.e. usable) version is planned for the end of Q2. I heavily use Yesod for the web stuff and pandoc-citeproc for dealing with citation formats. Deployment is handled with NixOS.

Further reading

http://pprjam.com/

5 Commercial Users

5.1 Well-Typed LLP

Report by: Adam Gundry
Participants: Duncan Coutts, Andres Löh and others

Well-Typed is a Haskell services company. We provide commercial support for Haskell as a development platform, including consulting services, training, bespoke software development and technical support. For more information, please take a look at our website or drop us an e-mail at ⟨info@well-typed.com⟩.

One of our main responsibilities is maintenance and release management for GHC (→ 3.1), thanks to support from several companies including Microsoft Research. Our full-time GHC team is led by Ben Gamari. If your company is interested in supporting work on GHC, either by contributing to the general maintenance pool or funding work on specific issues, please get in touch with us. We are also able to provide technical support with toolchain issues such as non-standard GHC configurations.

Through our work for IOHK we have recently been contributing significantly to the development of the Cardano cryptocurrency, which is implemented in Haskell. Cardano has been in the top 10 cryptocurrencies by market cap since December 2017. We are applying formal methods and related "semi-formal" techniques to achieve higher assurance and improved code quality in future iterations of the cryptocurrency (including everything from QuickCheck-compatible designs to developing formalized process calculi in Isabelle).

We endeavour to make our work available as open source software wherever possible, and contribute back to existing projects we use. Recently our contributions have included:

- Andres Löh and collaborators have proposed `DerivingVia`, a carefully-crafted extension to GHC's `GeneralizedNewtypeDeriving` feature that significantly increases the expressive power of Haskell's `deriving` construct.
- Duncan Coutts, Austin Seipp, Ben Gamari and other contributors have released the `cborg` and `serialise` libraries (formerly known as `binary-serialise-cbor`). The `serialise` library provides efficient serialisation of Haskell values as ByteStrings for storage or transmission purposes (as in the `binary` package). The underlying binary format is the standardised Concise Binary Object Representation (CBOR), which is both fast and capable of being inspected or analysed without custom tools.
- Edsko de Vries developed `visualize-cbn`, a tool for generating visualisations of lazy evaluation, including as interactive HTML/JavaScript pages. This is useful for teaching and for understanding the reduction behaviour of programs.
- Edsko also recently released `friendly`, a simple command-line tool for generic pretty-printing.
- Andres Löh continued development of the generic programming library `generics-sop` (→ 4.1.2).

We are looking forward to the next Haskell eXchange in London from 11–12th October 2018, and we are planning another free Hackathon on 13th October. Together with Skills Matter, we will be offering public Haskell training courses in London around the Haskell eXchange (8th–10th, 15th–16th October). Registration is now open for both the courses and eXchange, so if you would like to come, register now!

We are 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 drop us an e-mail.

Further reading

- Home page: https://www.well-typed.com/
- Blog: https://www.well-typed.com/blog/
- Training page: https://www.well-typed.com/services_training/
- Cardano: https://www.cardano.org/
- `DerivingVia proposal`: https://github.com/ghc-proposals/ghc-proposals/pull/120
- `cborg`: http://hackage.haskell.org/package/cborg
- `serialise`: http://hackage.haskell.org/package/serialise
- `generics-sop`: http://hackage.haskell.org/package/generics-sop
- `visualize-cbn`: http://hackage.haskell.org/package/visualize-cbn
- `friendly`: https://hackage.haskell.org/package/friendly
- Haskell eXchange 2018: https://skillsmatter.com/conferences/10237-haskell-exchange-2018
- Skills Matter courses: https://skillsmatter.com/explore?content=courses&location=London&q=haskell

5.2 Keera Studios LTD

Report by: Christina Zeller and Ivan Perez

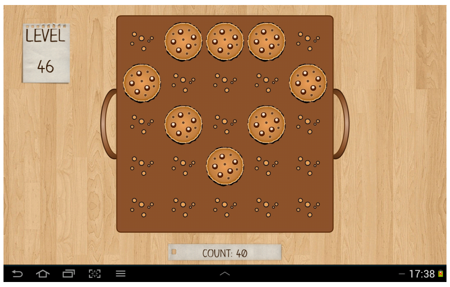
Keera Studios Ltd. is a Haskell development studio that focuses on games and mobile apps for both iOS and Android.

Last year, we completed a set of development tools for mobile Haskell games that enables compiling, testing, packaging and deploying mobile games with no effort. Our framework is versatile enough to accomodate not only games, but also mobile apps using standard widget toolkits available on mobile platforms, in order to provide a natural look and feel.

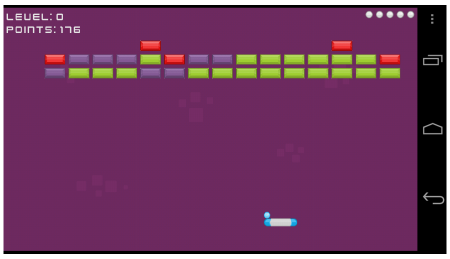
Our development toolkit if based on two tools, Andronaut and Curiosity, that provide, respectively, Android and iOS application templates and compile, package, sign and upload Haskell games and mobile apps to online stores. They have been designed to be trivial use, both during development and in CI servers: each tool downloads GHC pre-compiled for each target, as well as other SDKs and tools needed for mobile platforms. We use Travis CI to compile, package, sign and automatically upload our games to Google Play for Android (and tweet when a major release is out!). When developing locally, Andronaut and Curiosity maximize caching, and are able to recompile a mobile app and deploy it to a phone (connected via USB) in less than 45 seconds, or to the online store in less than one minute.

We top our development framework with a novel tool for testing and debugging mobile games: Haskell Titan^{TM} (Testing Infrastructure for Temporal AbstractioNs). Haskell Titan^{TM} is designed to take advantage of Haskell's referential transparency to deliver fully reproducible game runs that can be saved, replayed, paused, played backwards, modified and debugged. Our GUI tool communicates with the game running on a phone or a computer, and uses Temporal Logic to specify game assertions. Players can record a game run with minimal overhead and send it over the internet to our servers, which we can use to reproduce bugs deterministically on the same architecture. Our tool also integrates well with QuickCheck, which we use to test game assertions and find counterexamples. Effectively, we can see QuickCheck play.

We have published Magic Cookies!, the first commercial game for iOS and Android written in Haskell, available on iTunes (https://goo.gl/6gB6sb) and on Google Play^{TM} (https://goo.gl/cM1tD8). Currently holding a 4.9/5 star review, Magic Cookies! been in the Top 100 of paid board games on Google Play US, and Top 125 in the European Union, number 7 for paid braingames in Taiwan, number 15 in Italy, and Top 50 in 110 countries.



We have developed a breakout clone that runs on Windows, Linux, Mac, iOS, Android and Web (http://goo.gl/53pK2x), using *hardware acceleration* and *parallelism*. This game has been released on Google Play for Android, is distributed pre-compiled for Ubuntu via Launchpad, and can be played directly on the browser, compiled via GHCJS. The desktop version additionally supports Nintendo Wiimotes and Kinect. Thanks to many volunteer Haskellers, this game has seen updated documentation, better support for all OSs, and now includes 9 new levels.



We are currently developing three arcade games for both Android and iOS. Soon we will release the first version of a game internally called 'npuzzles', a sliding block puzzle game with classic levels, as well as levels with altered or novel rules designed to surprise, challenge and engage the player. This game is the first built from scratch using our SAGE infrastructure (detailed below), and serves as a well documented game-programming teaching source using Haskell, functional reactive programming (Yampa), SAGE, and our standard game structure, to show how easy and clear Haskell desktop and mobile games can be. The second game is Escape, and has been designed to challenge user's ability to track multiple objects moving simultaneously in different directions. The third one, codenamed pang-a-lambda, is inspired by the classic Super Pang and takes advantage of newer developments in the FRP implementation Yampa to include declarative physics simulations and time transformations, including the possibility of reversing time.

We are also producing a visual interactive game about the war in Ukraine, which depicts the everyday life of people affected by the war, in an effort to raise awareness about the current state of affairs, how people are being treated and what they have lost. This game invites the player to see the human side of people who remain *invisible* to their own society, only to become an unnamed shield standing at the conflict line. On the basis of this game, our company has been accepted in Facebook's acceleration programme for new apps and startups FbStart, and being granted thousands of dollars in digital goods and services for support and advertising.

This year we have completed the first version of our Simple Arcade Game Engine (SAGE), which hides tedious details from the developer. SAGE provides solutions for aspects common to many games, like asset management, 2D drawings, interactive widgets and positioning, and input devices (mouse, keyboard, touchscreen, accelerometers, wiimote and kinect), different game screens (splash screen, menus, level selection), preferences, saving and restoring the game state, and basic 2D physics and collisions. SAGE works for mobile and desktop and supports multiple backends, like SDL and SDL2, and hiding the details from the developer. The first game released with SAGE will be npuzzles, and we are currently porting other games to this new framework before their release.

In previous years we also 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, Android and iOS. This work has been described in the paper "GALE: A functional Graphic Adventure Library and Engine", presented at FARM 2017.

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 have developed a battery of Haskell mobile demos, covering SDL multimedia (including demos for multi-touch and accelerometers), communication with Java via C/C++, Facebook/Twitter status sharing, access to each mobile ecosystem (for instance, to use Android built-in shared preferences storage system, or for in-app payments), and use to native mobile widgets on Android and iOS.

All of this proves that Haskell truly is a viable option for *professional game and app development*, for both mobile and desktop. Our novel testing and debugging tools show that Haskell can, in certain respects, be more suitable than other languages for game and app programming, especially for (heterogeneous) mobile devices. In combination with our samples and our compilation, packaging and deployment tool, we have a complete, pain-free mobile app development suite.

Our GUI applications are created using Keera Hails, our open-source *reactive* programming library (http://git.io/vTvXg). Keera Hails provides integration with GTK+, network sockets, files, FRP Yampa signal functions and other external resources. Keera Hails is straightforward to adapt to new platforms, and we have shown reactive Haskell applications that work on iOS, Android, Windows, Linux, Mac and Web/DOM (using each platform's default widget system), just by choosing different backends (https://goo.gl/nFUA2u). We have used Keera Hails for our Graphic Adventure IDE, the open-source posture monitor Keera Posture (http://git.io/vTvXy), as well as multiple other commercial and open-source applications. Guerric Chupin (ENSTA ParisTech) and Henrik Nilsson (University of Nottingham (→ 6.6)) have also published Arpeggigon (→ 4.18.4) (https://gitlab.com/chupin/arpeggigon), an interactive cellular automaton for composing groove-based music, which combines the FRP implementation Yampa and Keera Hails. Their results have been written up in an application paper, *Funky Grooves: Declarative Programming of Full-Fledged Musical Applications*, presented at PADL 2017, and demonstrated at FARM 2017, the Haskell eXchange 2017, and Haskell in Leipzig 2017.

Videos and details of our work are published regularly on Facebook (https://fb.me/keerastudios), on Twitter (https://www.twitter.com/keerastudios), and on our company website (http://www.keera.co.uk). If you want to use Haskell in your next game, desktop or mobile application, or to receive more information, please contact us at ⟨keera@keera.co.uk⟩.

5.3 McMaster Computing and Software Outreach

Report by:	Christopher Anand
Status:	active

McMaster Computer Science 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, using Elm for the programming portion of our programming. Elm looks a lot like Haskell, but does not have user-definable type classes and is strict. We presented a paper about our tools and curriculum at TFPIE 2017, https://www.cs.kent.ac.uk/people/staff/sjt/TFPIE2017/TFPIE_2017/Programme.html.

We would like to thank Google for igniteCS funding, which allowed us to run seven-week workshops in schools in high-needs schools. Many of their contributions are in our Hall of Fame: http://outreach.mcmaster.ca/menu/fame.html.

6 Research and User Groups

6.1 DataHaskell

Report by:	Marco Zocca
Participants:	Nikita Tchayka, Mahdi Dibaiee, John Vial, Stefan Dresselhaus, Michał Gajda, and many others
Status:	Ongoing

The DataHaskell community was initiated in September 2016 as a gathering place for scientific computing, machine learning and data science practitioners and Haskell programmers; we observe a growing interest in using functional composition, domain-specific languages and type inference for implementing robust and reusable data processing pipelines.

DataHaskell revolves around a Gitter chatroom and a GitHub organization. The community is slowly but steadily growing; new interested people join the chatroom discussion (which now counts more than 230 unique users) every few days, and 20 or so are active on an average week.

We have a documentation page that serves both as a knowledge base of related Haskell packages and frameworks and to coordinate development, along with a package benchmarking repository.

After an informal survey we concluded that large part of our userbase seems to be lacking most:

- an IDE for exploratory data analysis,
- a generic 'data-frame' for fast import and manipulation of heterogeneous tabular data,
- a native numerical back-end.
- type providers for easy import of diverse data,
- Docker images pre-built to support big numerical libraries in IHaskell notebook environment, like ATLAS, TensorFlow, or GSL.

The notebook-IDE situation has improved, thanks for example to an updated iHaskell and the new, native Haskell.do editor. Dataframes are a more subtle topic, that require domain-specific optimizations, and we are also actively working on that after adopting the `analyze` package.

Some of us met in person at ZuriHac (Zürich) and ICFP 2017 (Oxford). An informal DataHaskell workshop took place on Saturday 9, 2017 at ICFP, during which a series of lightning talks showed various applications of the current state of the library ecosystem, e.g. rendering mathematics-heavy code on Hackage, high-performance numerical computing, probabilistic EDSLs and notebook usage for exploratory data analysis.

The workshop was well-received and we are evaluating various options for upcoming meetings, e.g. to be hosted at either functional programming or data science conferences.

We cherish the open and multidisciplinary nature of our community, and welcome all new users and contributions.

Further reading

http://datahaskell.org

6.2 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

6.3 Artificial Intelligence and Software Technology at Goethe-University Frankfurt

Report by:	Nils Dallmeyer
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 that operate on a meta-representation of the language expressions and transformations.

We therefore developed variants of unification on the meta-representation: An expressive variant that covers all the specifics of normal-order reduction rules. Others are extensions of nominal unification with built-in alpha-equivalence: one variant can deal with recursive lets, the second one permits variable-variables, and a third one permits context-variables and a meta-form of distinct-variable-conditions.

Improvements In recent research we analyzed whether program transformations are optimizations, i.e. whether they improve the time and/or space 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. We also developed a simulation-based method to validate time-improvements respecting the sharing structure. We showed for different transformations that they are space improvements and developed a tool for specific space analyses.

Concurrency We analyzed a higher-order functional language with concurrent threads, monadic IO, MVars and concurrent futures which models Concurrent Haskell and 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. Moreover we developed an algorithm that optimizes concurrent programs w.r.t. space and analyzed the impact on complexity of synchronization requirements.

Grammar based compression This research topic focuses on algorithms on grammar compressed data like strings, matrices, and terms. We implemented several algorithms as a Haskell library.

Further reading

http://www.ki.informatik.uni-frankfurt.de/research/HCAR.html

6.4 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. In September Joanna Sharrad joined us as a new PhD student to work with Meng Wang and Olaf Chitil on type error debugging. For her undergraduate work on delta debugging of type errors she already won the second place in the Student Research

Competition at ICFP 2017. 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 built the lightweight tracer and algorithmic debugger Hoed for real-world Haskell programs and successfully defended his PhD thesis. Olaf Chitil develops these ideas on tracing further, with a small prototype tracer HatLight for a subset of Haskell and the aim of developing the Haskell tracer Hat further. Dominic Orchard is working on coeffectful programming and applying verification in computational science. He also develops tools in Haskell for analysing, refactoring and verifying Fortran programs. Meng Wang is working on lenses, bidirectional transformation and property-based testing (QuickCheck). Together with last years visitor Colin Runciman from the University of York, Stefan Kahrs is working on minimising regular expressions, implemented in Haskell. Scott Owens is working on verified compilers for the (strict) functional language CakeML. Simon Thompson, Scott Owens, Hugo Férée and Reuben Rowe are working on an EPSRC project on trustworthy refactoring. They are studying refactoring for CakeML and OCaml, informed by their previous work for Haskell and Erlang.

We are always looking for more PhD students. We are particularly keen to recruit students interested in programming tools for verification, compilation, 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/
- Haskell: the craft of functional programming: http://www.haskellcraft.com
- Parsers and static analysis tools for Fortran code in Haskell https://github.com/camfort/fortran-src
- A refactoring and verification tool for Fortran code in Haskell https://github.com/camfort/camfort
- 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/

6.5 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. One example is the Haskino Project (→ 4.14.3).

Further reading

The Functional Programming Group: http://www.ittc.ku.edu/csdl/fpg

6.6 Functional Programming Laboratory at the University of Nottingham

Report by:	Jennifer Hackett, Martin Handley

The School of Computer Science at the University of Nottingham is home to the Functional Programming Laboratory, a research group focused on all theoretical and practical aspects of functional programming, together with related topics such as type theory and category theory. The lab is led by Thorsten Altenkirch and Graham Hutton, with Henrik Nilsson and Venanzio Capretta as academic staff. In addition, we currently have three post-doctoral researchers, eight postgraduate students and one intern.

Our interests are wide-ranging, including:

Mind the Gap: Reasoning about Efficiency

The Functional Programming Lab is currently home to an EPSRC grant on reasoning about the efficiency of programs. Jennifer Hackett and Graham Hutton are currently preparing a paper for ICFP that presents a cost-aware parametricity theorem that produces free theorems that can be used to reason about time costs, alongside another paper exploring an approach to reasoning about efficiency based on metric spaces. Also, Martin Handley and Graham Hutton have developed a tool that will assist programmers to reason inequationally about the cost behaviour of their programs. A paper on this tool has been submitted to the upcoming European Symposium on Programming.

Generalizing Monads and Applicative Functors

Jan Bracker and Henrik Nilsson are working on unifying generalizations of monads and applicative functors. Their work has produced the supermonads library (→ 4.1.3) which offers a unified way to work with these generalized notions in Haskell (details in the independent entry). In addition to the practical support of these notions they are also exploring the theoretical foundations of said notions in category theory. Future work may involve generalizing arrows as well.

Functional Reactive Programming

Functional Reactive Programming (FRP) remains an active part of our research. Henrik Nilsson and Ivan Perez have worked on monadic FRP, FRP extensions such as time transformations, property-based testing and debugging for FRP, Reactive Values and Relations (RVR). We have also worked on applications of these technologies; e.g., to games and music. In particular, together with summer interns Guerric Chupin and Jin Zhan, we developed the Arpeggigon (→ 4.18.4), a Functional Reactive Musical Automaton, as a substantial test case for FRP and RVR that is also an interesting and fun application in its own right. This has led to a number of publications and talks (PADL, ICFP, Haskell Symposium, FARM, London Haskell Meetup, Haskell eXchange, Haskell in Leipzig) as well as open source software releases. The Yampa (→ 4.17.1) FRP implementation is also actively being maintained.

Jonathan Thaler is applying Functional Reactive Programming to agent-based simulation (→ 4.17.3) with the aim to create more robust and correct implementations.

Teaching

The lab plays an active role in teaching, with Haskell playing an important role in many of the modules offered. Modules include: Programming, Programming Paradigms, Advanced Functional Programming, Machines and Their Languages, Compilers, Introduction to Formal Reasoning, Advanced Algorithms and Data Structures, Languages and Computation and Foundations of Programming.

Other Activities

Every Friday afternoon we hold the FP Café, an informal meeting where we hold whiteboard discussions on current activities, general business or anything that sounds interesting. Visitors are welcome: if you want to come along (perhaps even to give a talk yourself), please contact us!

The lab plays a leading role in the Midlands Graduate School in the Foundations of Computing Science.

Further reading

http://www.nottingham.ac.uk/research/groups/fp-lab/

6.7 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

6.8 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.

We usually have dinner first and then move on to have a talk. Topics vary quite a bit, from introductory to advanced, from theoretical to practical, and we have been looking at other languages such as Scala or dependently typed languages as well.

There are typically between 5 and 15 attendees, and we often get visitors from Munich and Nürnberg.

New members are always welcome, whether they are Haskell beginners or experts. If you are living in the area or are visiting, please join! Meetings are announced a few weeks in advance on our meetup page: http://www.meetup.com/Regensburg-Haskell-Meetup/.

6.9 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/

6.10 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

6.11 RuHaskell – the Russian-speaking haskellers community

Report by:	Yuriy Syrovetskiy
Status:	active

RuHaskell is the Russian-speaking community of haskellers. We have a website with Haskell-related articles, a podcast, a subreddit and some Gitter chats including one for novice haskellers specially. We also organize mini-conferences about twice a year in Moscow, Russia. The 4th mini-conference has taken place in the April, 2017.

Further reading

- Short info: https://wiki.haskell.org/RuHaskell
- Website: ruhaskell.org
- Gitter chats: /ruHaskell/home
- Twitter channel: @ruHaskell
- Subreddit: /r/ruhaskell

6.12 NY Haskell Users Group and Compose Conference

Report by:	Gershom Bazerman
Status:	ongoing

Since 2012 the NY Haskell Users Group has been hosting monthly Haskell talks and the occasional hackathon. Over fifteen-hundred members are registered on Meetup for the group, and talk attendence ranges between sixty to one hundred and twenty. NYHUG has also been organizing, on and off, beginner-oriented hangouts where people can assemble and study and learn together. And as of recently, NYHUG has also been the home base for organizing a Haskell Programming from First Principles study group, as well as an active Slack channel where ongoing discussion for the reading group takes place.

In 2015, the NY Haskell organizers launched the Compose Conference, which was held again in 2016, with a sibling "Compose::Melbourne" conference being held in 2016 as well. Compose is a cross-language conference for functional programmers, focused on strongly-typed functional languages such as Haskell, OCaml, F#, and SML. It aims to be both practical and educational, among other things providing opportunity for researchers to present the more applicable elements of their work to a wide audience of professional and hobbyist functional programmers. It is our hope to continue Compose and also to extend it to sibling conferences in other geographic areas as well sharing similar goals and format.

Further reading

- http://www.meetup.com/NY-Haskell/
- http://www.composeconference.org/

6.13 Japan Haskell User Group – Haskell-jp

Report by:	Yuji Yamamoto
Status:	active

Japan Haskell User Group (a.k.a Haskell-jp) is a Haskellers' community group in Japan, established at April 2017.

Since it began, we have tried various activities to help Haskellers discuss, collect, learn, and deliver any information related to Haskell.

Today we have two major updates since the last HCAR:

New logo voted

Thanks to Fumiaki Kinoshita (https://github.com/fumieval) and the voters in Haskell-jp, we got a brand-new, nice, and more Japanese logo. It's nicknamed "Sakulambda", after its motif, sakura and the lambda character.

Haskell-jp

Submitting a new article to the Haskell-jp blog

All you have to do is send a pull request to the Haskell-jp repository with your article written in Markdown format.

We will review the rendered article as well as its source code, and deploy it by merging on master.

These features are implemented by Circle CI's build artifacts and Travis CI's automatic deployment.

Further reading

- https://haskell.jp/blog
- https://haskell.jp/antenna
- https://wiki.haskell.jp/
- https://www.reddit.com/r/haskell_jp/
- https://haskell-jp.slack.com
- https://github.com/haskell-jp

6.14 Tokyo Haskell Meetup – Casual, English-speaking monthly meetings in Tokyo

Report by:	Yasuaki Kudo
Status:	active

Tokyo Haskell Meetup is a group that meets mostly every month in Tokyo. English is the dominant language and we enjoy casual and lively discussions of Haskell and Functional Languages, (and others including Philosophy, Politics, Business, etc.), while helping each other learn Haskell. Our members proficiency in Haskell is very diverse – from the beginners (such as myself) to the experts.

Further reading

https://www.meetup.com/Tokyo-Haskell-Meetup

6.15 Functional Programming at the Telkom University

Report by:	Wisnu Adi Nurcahyo

Functional programmers are rare to find in Indonesia, especially for Haskell where they are less than 30 from hundreds of thousands programmers that the country has.

I started a functional programming group at Telkom University. My goal is to create a great community of functional programmers starting from university.

Contact

⟨wisnu@nurcahyo.me⟩

6.16 Haskell Serbia

Report by:	Sasa Bogicevic
Status:	Active

Haskell-serbia is website for the Haskell user group in Serbia.

The idea is to organize meetups, write tutorials and generally spread the Haskell programming language in Serbia.

It is in active development and website is up and running.

There are plans to add fresh new look to website, customize user profile pages, write more tutorials and generally get more people involved.

Further reading

https://haskell-serbia.com