What I Wish I Knew
When Learning Haskell

Stephen Diehl (**@smdiehl**)

The source for all code is available here. If there are any errors or you think of a more illustrative example feel free to submit a pull request on Github.

This is the third draft of this document.

License

This code and text are dedicated to the public domain. You can copy, modify, distribute and perform the work, even for commercial purposes, all without asking permission.

Changelog

2.2

Sections that have had been added or seen large changes:

- Irrefutable Patterns
- Hackage
- Exhaustiveness
- Stacktraces
- Laziness
- Skolem Capture
- Foreign Function Pointers
- Attoparsec Parser
- Inline Cmm
- PrimMonad
- Specialization
- unbound-generics
- Editor Integration
- EKG
- Nix
- Haddock
- Monad Tutorials Commentary
- Monad Morphisms
- Corecursion
- Category
- Arrows
- Bifunctors
- ExceptT
- hint / mueval
- Roles
- Higher Kinds
- Kind Polymorphism
- Numeric Tower
- SAT Solvers
- Graph
- Sparks
- Threadscope
- Generic Parsers
- GHC Block Diagram
- GHC Debug Flags
- Core
- Inliner
- Unboxed Types
- Runtime Memory Representation
- ghc-heapview
- STG
- Worker/Wrapper
- Z-Encoding
- Cmm
- Runtime Optimizations
- RTS Profiling
- Algebraic Relations

Basics

Cabal

Cabal is the build system for Haskell, it also doubles as a package manager.

For example to install the parsec package from Hackage to our system invoke the install command:

```
$ cabal install parsec          # latest version
$ cabal install parsec==3.1.5   # exact version
```

The usual build invocation for Haskell packages is the following:

```
$ cabal get parsec    # fetch source
$ cd parsec-3.1.5

$ cabal configure
$ cabal build
$ cabal install
```

To update the package index from Hackage run:

```
$ cabal update
```

To start a new Haskell project run

```
$ cabal init
$ cabal configure
```

A `.cabal` file will be created with the configuration options for our new project.

The latest feature of Cabal is the addition of Sandboxes (in cabal > 1.18) which are self contained environments of Haskell packages separate from the global package index stored in the `./.cabal-sandbox` of our project's root. To create a new sandbox for our cabal project run.

```
$ cabal sandbox init
```

In addition the sandbox can be torn down.

```
$ cabal sandbox delete
```

Invoking the cabal commands when in the working directory of a project with a sandbox configuration set up alters the behavior of cabal itself. For example the `cabal install` command will only alter the install to the local package index and will not touch the global configuration.

To install the dependencies from the cabal file into the newly created sandbox run:

```
$ cabal install --only-dependencies
```

Dependencies can also be built in parallel by passing `-j<n>` where `n` is the number of concurrent builds.

```
$ cabal install -j4 --only-dependencies
```

Let's look at an example cabal file, there are two main entry points that any package may provide: a `library` and an `executable`. Multiple executables can be defined, but only one library. In addition there is a special form of executable entry point `Test-Suite` which defines an interface for unit tests to be invoked from cabal.

For a library the `exposed-modules` field in the cabal file indicates which modules within the package structure will be publicly visible when the package is installed, these are the user-facing APIs that we wish to expose to downstream consumers.

For an executable the `main-is` field indicates the Main module for the project that exports the `main` function to run for the executable logic of the application.

```
name:             mylibrary
version:          0.1
cabal-version:    >= 1.10
author:           Paul Atreides
license:          MIT
license-file:     LICENSE
synopsis:         The code must flow.
category:         Math
tested-with:      GHC
build-type:       Simple

library
    exposed-modules:
      Library.ExampleModule1
      Library.ExampleModule2

    build-depends:
      base >= 4 && < 5

    default-language: Haskell2010

    ghc-options: -O2 -Wall -fwarn-tabs

executable "example"
    build-depends:
        base >= 4 && < 5,
        mylibrary == 0.1
    default-language: Haskell2010
    main-is: Main.hs

Test-Suite test
   type: exitcode-stdio-1.0
   main-is: Test.hs
   default-language: Haskell2010
```

```
build-depends:
    base >= 4 && < 5,
    mylibrary == 0.1
```

To run the "executable" for a library under the cabal sandbox:

```
$ cabal run
$ cabal run <name>
```

To load the "library" into a GHCi shell under the cabal sandbox:

```
$ cabal repl
$ cabal repl <name>
```

The `<name>` metavariable is either one of the executable or library declarations in the cabal file, and can optionally be disambiguated by the prefix `exe:<name>` or `lib:<name>` respectively.

To build the package locally into the `./dist/build` folder execute the `build` command.

```
$ cabal build
```

To run the tests, our package must itself be reconfigured with the `--enable-tests` and the `build-depends` from the Test-Suite must be manually installed if not already.

```
$ cabal install --only-dependencies --enable-tests
$ cabal configure --enable-tests
$ cabal test
$ cabal test <name>
```

In addition arbitrary shell commands can also be invoked with the GHC environmental variables set up for the sandbox. Quite common is to invoke a new shell with this command such that the `ghc` and `ghci` commands use the sandbox (they don't by default, which is a common source of frustration).

```
$ cabal exec
$ cabal exec sh # launch a shell with GHC sandbox path set.
```

The haddock documentation can be built for the local project by executing the `haddock` command, it will be built to the `./dist` folder.

```
$ cabal haddock
```

When we're finally ready to upload to Hackage (presuming we have a Hackage account set up), then we can build the tarball and upload with the following commands:

```
$ cabal sdist
$ cabal upload dist/mylibrary-0.1.tar.gz
```

Sometimes you'd also like to add a library from a local project into a sandbox. In this case the add-source command can be used to bring it into the sandbox from a local directory.

```
$ cabal sandbox add-source /path/to/project
```

The current state of a sandbox can be frozen with all current package constraints enumerated.

```
$ cabal freeze
```

This will create a file `cabal.config` with the constraint set.

```
constraints: mtl ==2.2.1,
             text ==1.1.1.3,
             transformers ==0.4.1.0
```

Using the `cabal repl` and `cabal run` commands is preferable but sometimes we'd like to manually perform their equivalents at the shell, there are several useful aliases that rely on shell directory expansion to find the package database in the current working directory and launch GHC with the appropriate flags:

```
alias ghc-sandbox="ghc -no-user-package-db -package-db .cabal-sandbox/*-packages.conf.
alias ghci-sandbox="ghci -no-user-package-db -package-db .cabal-sandbox/*-packages.cor
alias runhaskell-sandbox="runhaskell -no-user-package-db -package-db .cabal-sandbox/*-
```

There is also a zsh script to show the sandbox status of the current working directory in our shell.

```
function cabal_sandbox_info() {
    cabal_files=(*.cabal(N))
    if [ $#cabal_files -gt 0 ]; then
        if [ -f cabal.sandbox.config ]; then
            echo "%{$fg[green]%}sandboxed%{$reset_color%}"
        else
            echo "%{$fg[red]%}not sandboxed%{$reset_color%}"
        fi
    fi
```

```
  }

RPROMPT="\$(cabal_sandbox_info) $RPROMPT"
```

The cabal configuration is stored in `$HOME/.cabal/config` and contains various options including credential information for Hackage upload. One addition to configuration is to completely disallow the installation of packages outside of sandboxes to prevent accidental collisions.

```
-- Don't allow global install of packages.
require-sandbox: True
```

A library can also be compiled with runtime profiling information enabled. More on this is discussed in the section on Concurrency and profiling.

```
library-profiling: True
```

Another common flag to enable is the `documentation` which forces the local build of Haddock documentation, which can be useful for offline reference. On a Linux filesystem these are built to the `/usr/share/doc/ghc/html/libraries/` directory.

```
documentation: True
```

If GHC is currently installed the documentation for the Prelude and Base libraries should be available at this local link:

/usr/share/doc/ghc/html/libraries/index.html

See:

- An Introduction to Cabal Sandboxes
- Storage and Identification of Cabalized Packages

Hackage

Hackage is the canonical source of open source Haskell packages. Being a transitional language, Hackage is many things to many people but there seem to be two dominant philosophies:

Reusable Code / Building Blocks

Libraries exist as stable, community supported, building blocks for building higher level functionality on top of an edifice which is common and stable. The author(s) of the library have written the library as a means of packaging up their understanding of a problem domain so that others can build on their understanding and expertise.

A Staging Area / Request for Comments

A common philosophy is that Hackage is a place to upload experimental libraries up as a means of getting community feedback and making the code publicly available. The library author(s) often rationalize putting these kind of libraries up undocumented, often not indicating what the library even does, by simply stating that they intend to tear it all down and rewrite it later. This unfortunately means a lot of Hackage namespace has become polluted with dead-end bit-rotting code.

Many other language ecosystems (Python, NodeJS, Ruby) favor the former philosophy, and coming to Haskell can be kind of unnerving to see *thousands of libraries without the slightest hint of documentation or description of purpose*. It is an open question about the cultural differences between the two philosophies and how sustainable the current cultural state of Hackage is.

Needless to say there is a lot of very low-quality Haskell code and documentation out there today, and being conservative in library assessment is a necessary skill.

As a rule of thumb if the Haddock docs for the library does not have a **minimal worked example**, it is usually safe to assume that it is a RFC-style library and probably should be avoided.

There are several authors who it is usually safe to assume have uploaded a stable and usable library. These include, but are not limited to:

- Bryan O'Sullivan
- Johan Tibell
- Simon Marlow
- Gabriel Gonzalez
- Roman Leshchinskiy

GHCi

GHCi is the interactive shell for the GHC compiler. GHCi is where we will spend most of our time.

Command	Shortcut	Action
:reload	:r	Code reload
:type	:t	Type inspection
:kind	:k	Kind inspection
:info	:i	Information
:print	:p	Print the expression
:edit	:e	Load file in system editor.

The introspection commands are an essential part of debugging and interacting with Haskell code:

```
λ: :type 3
3 :: Num a => a
```

```
λ: :kind Either
Either :: * -> * -> *
```

```
λ: :info Functor
class Functor f where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
        -- Defined in 'GHC.Base'
  ...
```

```
λ: :i (:)
data [] a = ... | a : [a]       -- Defined in 'GHC.Types'
infixr 5 :
```

The current state of the global environment in the shell can also be queried. Such as module-level bindings and types:

```
λ: :browse
λ: :show bindings
```

Or module level imports:

```
λ: :show imports
import Prelude -- implicit
import Data.Eq
import Control.Monad
```

Or compiler-level flags and pragmas:

```
λ: :set
options currently set: none.
base language is: Haskell2010
with the following modifiers:
  -XNoDatatypeContexts
  -XNondecreasingIndentation
GHCi-specific dynamic flag settings:
other dynamic, non-language, flag settings:
  -fimplicit-import-qualified
warning settings:

λ: :showi language
base language is: Haskell2010
with the following modifiers:
  -XNoDatatypeContexts
  -XNondecreasingIndentation
  -XExtendedDefaultRules
```

Language extensions and compiler pragmas can be set at the prompt. See the Flag Reference for the vast set of compiler flag options. For example several common ones are:

```
:set -XNoMonomorphismRestriction
:set -fno-warn-unused-do-bind
```

Several commands for interactive options have shortcuts:

	Function
+t	Show types of evaluated expressions
+s	Show timing and memory usage

(+m) Enable multi-line expression delimited by (:{) and (:}) .

```
λ: :set +t
λ: []
[]
it :: [a]
```

```
λ: :set +s
λ: foldr (+) 0 [1..25]
325
it :: Prelude.Integer
(0.02 secs, 4900952 bytes)
```

```
λ: :{
λ:| let foo = do
λ:|          putStrLn "hello ghci"
λ:| :}
λ: foo
"hello ghci"
```

The configuration for the GHCi shell can be customized globally by defining a (ghci.conf) in (\$HOME/.ghc/) or in the current working directory as (./.ghci.conf) .

For example we can add a command to use the Hoogle type search from within GHCi.

```
cabal install hoogle
```

We can use it by adding a command to our (ghci.conf) .

```
:set prompt "λ: "

:def hlint const . return $ ":! hlint \"src\""
:def hoogle \s -> return $ ":! hoogle --count=15 \"" ++ s ++ "\""
```

```
λ: :hoogle (a -> b) -> f a -> f b
Data.Traversable fmapDefault :: Traversable t => (a -> b) -> t a -> t b
Prelude fmap :: Functor f => (a -> b) -> f a -> f b
```

For reasons of sexiness it is desirable to set your GHC prompt to a (λ) or a (Π Σ) if you're into that lifestyle.

```
:set prompt "λ: "
:set prompt "ΠΣ: "
```

Editor Integration

Haskell has a variety of editor tools that can be used to provide interactive development feedback and functionality such as querying types of subexpressions, linting, type checking, and code completion.

Many prepackaged setups exist to expedite the process of setting up many of the programmer editors for Haskell development:

Vim

https://github.com/begriffs/haskell-vim-now

Emacs

https://github.com/chrisdone/emacs-haskell-config

The tools that many of these packages use behind the hood are usually available on cabal.

```
cabal install hdevtools
cabal install ghc-mod
cabal install hlint
cabal install ghcid
cabal install ghci-ng
```

In particular both `ghc-mod` and hdevtools can remarkably improve the efficiency and productivity.

See:

- A Vim + Haskell Workflow

Bottoms

```
error :: String -> a
undefined :: a
```

The bottom is a singular value that inhabits every type. When evaluated the semantics of Haskell no longer yields a meaningful value. It's usually written as the symbol ⊥ (i.e. the compiler flipping you off).

An example of an infinite looping term:

```
f :: a
f = let x = x in x
```

The `undefined` function is nevertheless extremely practical to accommodate writing incomplete programs and for debugging.

```
f :: a -> Complicated Type
f = undefined -- write tomorrow, typecheck today!
```

Partial functions from non-exhaustive pattern matching is probably the most common introduction of bottoms.

```
data F = A | B
case x of
  A -> ()
```

The above is translated into the following GHC Core with the exception inserted for the non-exhaustive patterns. GHC can be made more vocal about incomplete patterns using the `-fwarn-incomplete-patterns` and `-fwarn-incomplete-uni-patterns` flags.

```
case x of _ {
  A -> ();
  B -> patError "<interactive>:3:11-31|case"
}
```

The same holds with record construction with missing fields, although there's almost never a good reason to construct a record with missing fields and GHC will warn us by default.

```
data Foo = Foo { example1 :: Int }
f = Foo {}
```

Again this has an error term put in place by the compiler:

```
Foo (recConError "<interactive>:4:9-12|a")
```

What's not immediately apparent is that they are used extensively throughout the Prelude, some for practical reasons others for historical reasons. The canonical example is the `head` function which as written `[a] -> a` could not be well-typed without the bottom.

```
import GHC.Err
```

```haskell
import Prelude hiding (head, (!!), undefined)

-- degenerate functions

undefined :: a
undefined =  error "Prelude.undefined"

head :: [a] -> a
head (x:_) =  x
head []    =  error "Prelude.head: empty list"

(!!) :: [a] -> Int -> a
xs      !! n | n < 0 =  error "Prelude.!!: negative index"
[]      !! _         =  error "Prelude.!!: index too large"
(x:_)   !! 0         =  x
(_:xs)  !! n         =  xs !! (n-1)
```

It's rare to see these partial functions thrown around carelessly in production code and the preferred method is instead to use the safe variants provided in (Data.Maybe) combined with the usual fold functions (maybe) and (either) or to use pattern matching.

```haskell
listToMaybe :: [a] -> Maybe a
listToMaybe []     =  Nothing
listToMaybe (a:_)  =  Just a
```

When a bottom defined in terms of error is invoked it typically will not generate any position information, but the function used to provide assertions (assert) can be short circuited to generate position information in the place of either (undefined) or (error) call.

```haskell
import GHC.Base

foo :: a
foo = undefined
-- *** Exception: Prelude.undefined

bar :: a
bar = assert False undefined
-- *** Exception: src/fail.hs:8:7-12: Assertion failed
```

See: Avoiding Partial Functions

Exhaustiveness

Pattern matching in Haskell allows for the possibility of non-exhaustive patterns, or cases which are not exhaustive and instead of yielding a value diverge.

Partial functions from non-exhaustivity are controversial subject, and large use of non-exhaustive patterns is considered a dangerous code smell. Although the complete removal of non-exhaustive patterns from the language entirely would itself be too restrictive and forbid too many valid programs.

For example, the following function given a Nothing will crash at runtime and is otherwise a valid type-checked program.

```
unsafe (Just x) = x + 1
```

There are however flags we can pass to the compiler to warn us about such things or forbid them entirely either locally or globally.

```
$ ghc -c -Wall -Werror A.hs
A.hs:3:1:
    Warning: Pattern match(es) are non-exhaustive
             In an equation for 'unsafe': Patterns not matched: Nothing
```

The `-Wall` or incomplete pattern flag can also be added on a per-module basis with the `OPTIONS_GHC` pragma.

```
{-# OPTIONS_GHC -Wall #-}
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}
```

A more subtle case is when implicitly pattern matching with a single "uni-pattern" in a lambda expression. The following will fail when given a Nothing.

```
boom = \(Just a) -> something
```

This occurs frequently in let or do-blocks which after desugaring translate into a lambda like the above example.

```
boom = let
  Just a = something

boom = do
  Just a <- something
```

GHC can warn about these cases with the `-fwarn-incomplete-uni-patterns` flag.

Grossly speaking any non-trivial program will use some measure of partial functions, it's simply a fact. This just means there exists obligations for the programmer than cannot be manifest in the Haskell type system. Although future projects like LiquidHaskell may potentially offer a way to overcome this with more sophisticated refinement types, this is an open research problem though.

Debugger

Although its use is somewhat rare, GHCi actually does have a builtin debugger. Debugging uncaught exceptions from bottoms or asynchronous exceptions is in similar style to debugging segfaults with gdb.

```
λ: :set -fbreak-on-exception
λ: :trace main
λ: :hist
λ: :back
```

Stacktraces

With runtime profiling enabled GHC can also print a stack trace when an diverging bottom term (error, undefined) is hit, though this requires a special flag and profiling to be enabled, both are disabled by default. So for example:

```
import Control.Exception

f x = g x

g x = error (show x)

main = try (evaluate (f ())) :: IO (Either SomeException ())
```

```
$ ghc -O0 -rtsopts=all -prof -auto-all --make stacktrace.hs
./stacktrace +RTS -xc
```

And indeed the runtime tells us that the exception occurred in the function ⓖ and enumerates the call stack.

```
*** Exception (reporting due to +RTS -xc): (THUNK_2_0), stack trace:
  Main.g,
  called from Main.f,
  called from Main.main,
  called from Main.CAF
  --> evaluated by: Main.main,
  called from Main.CAF
```

It is best to run this without optimizations applied ⌐-O0⌐ so as to preserve the original call stack as represented in the source. With optimizations applied this may often entirely different since GHC will rearrange the program in rather drastic ways.

See:

- xc flag

Trace

Haskell being pure has the unique property that most code is introspectable on its own, as such the "printf" style of debugging is often unnecessary when we can simply open GHCi and test the function. Nevertheless Haskell does come with an unsafe ⌐trace⌐ function which can be used to perform arbitrary print statements outside of the IO monad.

```haskell
import Debug.Trace

example1 :: Int
example1 = trace "impure print" 1

example2 :: Int
example2 = traceShow "tracing" 2

example3 :: [Int]
example3 = [trace "will not be called" 3]

main :: IO ()
main = do
  print example1
  print example2
  print $ length example3
-- impure print
-- 1
-- "tracing"
-- 2
-- 1
```

The function itself is impure (it uses `unsafePerformIO` under the hood) and shouldn't be used in stable code.

In addition to just the trace function, several common monadic patterns are quite common.

```haskell
import Text.Printf
import Debug.Trace

traceM :: (Monad m) => String -> m ()
traceM string = trace string $ return ()

traceShowM :: (Show a, Monad m) => a -> m ()
traceShowM = traceM . show

tracePrintfM :: (Monad m, PrintfArg a) => String -> a -> m ()
tracePrintfM s = traceM . printf s
```

Typed Holes

Since GHC 7.8 we have a new tool for debugging incomplete programs by means of *typed holes*. By placing an underscore on any value on the right hand-side of a declaration GHC will throw an error during type-checker that reflects the possible values that could placed at this point in the program to make the program type-check.

```haskell
instance Functor [] where
  fmap f (x:xs) = f x : fmap f _
```

```
[1 of 1] Compiling Main             ( src/typedhole.hs, interpreted )

src/typedhole.hs:7:32:
    Found hole '_' with type: [a]
    Where: 'a' is a rigid type variable bound by
              the type signature for fmap :: (a -> b) -> [a] -> [b]
              at src/typedhole.hs:7:3
    Relevant bindings include
      xs :: [a] (bound at src/typedhole.hs:7:13)
      x :: a (bound at src/typedhole.hs:7:11)
      f :: a -> b (bound at src/typedhole.hs:7:8)
      fmap :: (a -> b) -> [a] -> [b] (bound at src/typedhole.hs:7:3)
    In the second argument of 'fmap', namely '_'
    In the second argument of '(:)', namely 'fmap f _'
    In the expression: f x : fmap f _
Failed, modules loaded: none.
```

GHC has rightly suggested that the expression needed to finish the program is `xs :: [a]`.

Nix

Nix is a package management system with a larger scope than cabal. It is generally not a Haskell specific project although much work has been done to integrate it with the existing cabal infrastructure. *Nix is not a replacement for cabal* but can be used to subsume some of cabal's work by building up isolated development environments that can include Haskell libraries (installed from binary packages) and arbitrary system libraries that can be linked into compiled Haskell programs.

Use of Nix is somewhat controversial in some aspects since it requires us to buy into a much larger system and write an additional set of configuration files in an entirely different Nix specification language. It is unclear what the future of Haskell and Nix will be and whether it is a workaround around some current cabal pain points or a deeper unifying model.

Once the NixOS package manager is installed you can start a nix shell on the fly with a bunch of packages installed in the nixos repos.

```
$ nix-shell -p haskellPackages.parsec -p haskellPackages.mtl --command ghci
```

This is of course not limited to haskell packages, and there is a wide variety of binary packages and libraries available. If your library depends on a specific version of GNU readline, Nix can for example manage this dependency while system libraries are outside the scope of `cabal-install`.

```
$ nix-shell -p llvm -p julia -p emacs
```

The Nix workflow for Haskell consists of a sequence like the following:

```
$ cabal init
... usual setup ...
$ cabal2nix mylibrary.cabal --sha256=0 > shell.nix
```

This will generate a file like the following:

```
# This file was auto-generated by cabal2nix. Please do NOT edit manually!

{ cabal, mtl, transformers
}:

cabal.mkDerivation (self: {
  pname = "mylibrary";
  version = "0.1.0.0";
  sha256 = "0";
  isLibrary = true;
  isExecutable = true;
  buildDepends = [
    mtl transformers
  ];
})
```

We'll need to manually edit the file:

```
# This file was auto-generated by cabal2nix. Please do NOT edit manually!

{ haskellPackages ? (import <nixpkgs> {}).haskellPackages }:

haskellPackages.cabal.mkDerivation (self: {
  pname = "mylibrary";
  version = "0.1.0.0";
  src = "./.";
  isLibrary = true;
  isExecutable = true;
  buildDepends = with haskellPackages; [
    mtl transformers
  ];
  buildTools = with haskellPackages; [ cabalInstall ];
})
```

There you go, now you can launch the cabal repl for your project with:

```
$ nix-shell --command "cabal repl"
```

This process has been automated by another library called cabal2nix4dev:

See:

- cabal2nix4dev

Haddock

Haddock is the automatic documentation tool for Haskell source code. It integrates with the usual cabal toolchain.

```haskell
-- | Documentation for f
f :: a -> a
f = ...
```

```haskell
-- | Multiline documentation for the function
-- f with multiple arguments.
fmap :: Functor f =>
    => (a -> b)    -- ^ function
    -> f a         -- ^ input
    -> f b         -- ^ output
```

```haskell
data T a b
  = A a -- ^ Documentation for A
  | B b -- ^ Documentation for B
```

Elements within a module (value, types, classes) can be hyperlinked by enclosing the identifier in single quotes.

```haskell
data T a b
  = A a -- ^ Documentation for 'A'
  | B b -- ^ Documentation for 'B'
```

Modules themselves can be referenced by enclosing them in double quotes.

```haskell
-- | Here we use the "Data.Text" library and import
-- the 'Data.Text.pack' function.
```

```haskell
-- | An example of a code block.
--
-- @
--     f x = f (f x)
-- @
--
-- > f x = f (f x)
```

```haskell
-- | Example of an interactive shell session.
```

```
--
-- >>> factorial 5
-- 120
```

Headers for specific blocks can be added by prefacing the comment in the module block with a star:

```
module Foo (
  -- * My Header
  example1,
  example2
)
```

Sections can also be delineated by $ blocks that pertain to references in the body of the module:

```
module Foo (
  -- $section1
  example1,
  example2
)

-- $section1
-- Here is the documentation section that describes the symbols
-- 'example1' and 'example2'.
```

Links can be added with the syntax:

```
<url text>
```

Images can can also be included, so long as the path is relative to the haddock or an absolute reference.

```
<<diagram.png title>>
```

Haddock options can also be specified with pragmas in the source, either on module or project level.

```
{-# OPTIONS_HADDOCK show-extensions, ignore-exports #-}
```

Option	Description
ignore-exports	Ignores the export list and includes all signatures in scope.
not-home	Module will not be considered in the root documentation.
show-extensions	Annotates the documentation with the language extensions used.
hide	Forces the module to be hidden from Haddock.

Monads

Eightfold Path to Monad Satori

Much ink has been spilled waxing lyrical about the supposed mystique of monads. Instead I suggest a path to enlightenment:

1. Don't read the monad tutorials.
2. No really, don't read the monad tutorials.
3. Learn about Haskell types.
4. Learn what a typeclass is.
5. Read the Typeclassopedia.
6. Read the monad definitions.
7. Use monads in real code.
8. Don't write monad-analogy tutorials.

In other words, the only path to understanding monads is to read the fine source, fire up GHC and write some code. Analogies and metaphors will not lead to understanding.

Monadic Myths

The following are all **false**:

- Monads are impure.
- Monads are about effects.
- Monads are about state.
- Monads are about imperative sequencing.
- Monads are about IO.
- Monads are dependent on laziness.
- Monads are a "back-door" in the language to perform side-effects.
- Monads are an embedded imperative language inside Haskell.
- Monads require knowing abstract mathematics.

See: What a Monad Is Not

Laws

Monads are not complicated, the implementation is a typeclass with two functions, `(>>=)` pronounced "bind" and `return`. Any preconceptions one might have for the word "return" should be discarded, it has an entirely different meaning.

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Together with three laws that all monad instances must satisfy.

Law 1

```
return a >>= f ≡ f a
```

Law 2

```
m >>= return ≡ m
```

Law 3

```
(m >>= f) >>= g ≡ m >>= (\x -> f x >>= g)
```

There is an auxiliary function ((>>)) defined in terms of the bind operation that discards its argument.

```
(>>) :: Monad m => m a -> m b -> m b
m >> k = m >>= \_ -> k
```

See: Monad Laws

Do Notation

Monads syntax in Haskell is written in sugared form that is entirely equivalent to just applications of the monad operations. The desugaring is defined recursively by the rules:

```
do { a <- f ; m } ≡ f >>= \a -> do { m }
do { f ; m } ≡ f >> do { m }
do { m } ≡ m
```

So for example the following are equivalent:

```
do
  a <- f
  b <- g
  c <- h
  return (a, b, c)

do {
  a <- f ;
  b <- g ;
  c <- h ;
  return (a, b, c)
}

f >>= \a ->
  g >>= \b ->
    h >>= \c ->
      return (a, b, c)
```

If one were to write the bind operator as an uncurried function (this is not how Haskell uses it) the same desugaring might look something like the following chain of nested binds with lambdas.

```
bindMonad(f, lambda a:
  bindMonad(g, lambda b:
    bindMonad(h, lambda c:
      returnMonad (a,b,c))))
```

In the do-notation the monad laws from above are equivalently written:

Law 1

```
  do x <- m
     return x

= do m
```

Law 2

```
  do y <- return x
     f y

= do f x
```

Law 3

```
  do b <- do a <- m
              f a
     g b

= do a <- m
     b <- f a
     g b

= do a <- m
     do b <- f a
        g b
```

See: <u>Haskell 2010: Do Expressions</u>

Maybe

The *Maybe* monad is the simplest first example of a monad instance. The Maybe monad models computations which fail to yield a value at any point during computation.

```
data Maybe a = Just a | Nothing
```

```haskell
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= k = Nothing

  return = Just
```

```haskell
(Just 3) >>= (\x -> return (x + 1))
-- Just 4

Nothing >>= (\x -> return (x + 1))
-- Nothing

return 4 :: Maybe Int
-- Just 4
```

```haskell
example1 :: Maybe Int
example1 = do
  a <- Just 3
  b <- Just 4
  return $ a + b
-- Just 7

example2 :: Maybe Int
example2 = do
  a <- Just 3
  b <- Nothing
  return $ a + b
-- Nothing
```

List

The *List* monad is the second simplest example of a monad instance.

```haskell
instance Monad [] where
  m >>= f   =  concat (map f m)
  return x  =  [x]
```

So for example with:

```haskell
m = [1,2,3,4]
f = \x -> [1,0]
```

The evaluation proceeds as follows:

```
m >>= f
==> [1,2,3,4] >>= \x -> [1,0]
==> concat (map (\x -> [1,0]) [1,2,3,4])
==> concat ([[1,0],[1,0],[1,0],[1,0]])
==> [1,0,1,0,1,0,1,0]
```

The list comprehension syntax in Haskell can be implemented in terms of the list monad.

```
a = [f x y | x <- xs, y <- ys, x == y ]

-- Identical to `a`
b = do
  x <- xs
  y <- ys
  guard $ x == y
  return $ f x y
```

```
example :: [(Int, Int)]
example = do
  a <- [1,2]
  b <- [10,20]
  return (a,b)

example' :: [(Int, Int)]
example' =
  [1,2] >>= \a ->
    [10,20] >>= \b ->
      return (a,b)

example'' :: [(Int, Int)]
example'' =
  concat (map (\a -> concat (map (\b -> return (a,b)) [10,20])) [1,2])

main = return ()
```

IO

A value of type `IO a` is a computation which, when performed, does some I/O before returning a value of type `a`. Desugaring the IO monad:

```
main :: IO ()
main = do putStrLn "What is your name: "
          name <- getLine
          putStrLn name
```

```
main :: IO ()
main = putStrLn "What is your name:" >>=
       \_    -> getLine >>=
       \name -> putStrLn name
```

```
main :: IO ()
main = putStrLn "What is your name: " >> (getLine >>= (\name -> putStrLn name))
```

See: Haskell 2010: Basic/Input Output

Whats the point?

Consider the non-intuitive fact that we now have a uniform interface for talking about three very different but foundational ideas for programming: *Failure*, *Collections*, and *Effects*.

Let's write down a new function called `sequence` which folds a function `mcons`, which we can think of as analogues to the list constructor (i.e. `(a : b : [])`) except it pulls the two list elements out of two monadic values (`p`, `q`) using bind.

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr mcons (return [])

mcons :: Monad m => m t -> m [t] -> m [t]
mcons p q = do
  x <- p
  y <- q
  return (x:y)
```

What does this function mean in terms of each of the monads discussed above?

Maybe

Sequencing a list of a `Maybe` values allows us to collect the results of a series of computations which can possibly fail and yield the aggregated values only if they all succeeded.

```
sequence :: [Maybe a] -> Maybe [a]
```

```
sequence [Just 3, Just 4]
-- Just [3,4]
sequence [Just 3, Just 4, Nothing]
-- Nothing
```

List

Since the bind operation for the list monad forms the pairwise list of elements from the two operands,

folding the bind over a list of lists with `sequence` implements the general Cartesian product for an arbitrary number of lists.

```
sequence :: [[a]] -> [[a]]
```

```
sequence [[1,2,3],[10,20,30]]
-- [[1,10],[1,20],[1,30],[2,10],[2,20],[2,30],[3,10],[3,20],[3,30]]
```

IO

Sequence takes a list of IO actions, performs them sequentially, and returns the list of resulting values in the order sequenced.

```
sequence :: [IO a] -> IO [a]
```

```
sequence [getLine, getLine]
-- a
-- b
-- ["a","b"]
```

So there we have it, three fundamental concepts of computation that are normally defined independently of each other actually all share this similar structure that can be abstracted out and reused to build higher abstractions that work for all current and future implementations. If you want a motivating reason for understanding monads, this is it! This is the essence of what I wish I knew about monads looking back.

See: Control.Monad

Reader Monad

The reader monad lets us access shared immutable state within a monadic context.

```
ask :: Reader r a
asks :: (r -> a) -> Reader r a
local :: (r -> b) -> Reader b a -> Reader r a
runReader :: Reader r a -> r -> a
```

```
import Control.Monad.Reader

data MyContext = MyContext
  { foo :: String
  , bar :: Int
  } deriving (Show)
```

```haskell
computation :: Reader MyContext (Maybe String)
computation = do
  n <- asks bar
  x <- asks foo
  if n > 0
    then return (Just x)
    else return Nothing

ex1 :: Maybe String
ex1 = runReader computation $ MyContext "hello" 1

ex2 :: Maybe String
ex2 = runReader computation $ MyContext "haskell" 0
```

A simple implementation of the Reader monad:

```haskell
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad (Reader r) where
  return a = Reader $ \_ -> a
  m >>= k  = Reader $ \r -> runReader (k (runReader m r)) r

ask :: Reader a a
ask = Reader id

asks :: (r -> a) -> Reader r a
asks f = Reader f

local :: (r -> b) -> Reader b a -> Reader r a
local f m = Reader $ runReader m . f
```

Writer Monad

The writer monad lets us emit a lazy stream of values from within a monadic context.

```haskell
tell :: w -> Writer w ()
execWriter :: Writer w a -> w
runWriter :: Writer w a -> (a, w)
```

```haskell
import Control.Monad.Writer

type MyWriter = Writer [Int] String

example :: MyWriter
example  = do
  tell [1..5]
  tell [5..10]
```

```
    return "foo"

output :: (String, [Int])
output = runWriter example
```

A simple implementation of the Writer monad:

```
import Data.Monoid

newtype Writer w a = Writer { runWriter :: (a, w) }

instance Monoid w => Monad (Writer w) where
  return a = Writer (a, mempty)
  m >>= k  = Writer $ let
      (a, w)  = runWriter m
      (b, w') = runWriter (k a)
      in (b, w `mappend` w')

execWriter :: Writer w a -> w
execWriter m = snd (runWriter m)

tell :: w -> Writer w ()
tell w = Writer ((), w)
```

This implementation is lazy so some care must be taken that one actually wants to only generate a stream of thunks. Often times it is desirable to produce a computation which requires a stream of thunks that can be pulled out of the `runWriter` lazily, but often times the requirement is to produce a finite stream of values that are forced at the invocation of `runWriter`. Undesired laziness from Writer is a common source of grief, but is very remediable.

State Monad

The state monad allows functions within a stateful monadic context to access and modify shared state.

```
runState  :: State s a -> s -> (a, s)
evalState :: State s a -> s -> a
execState :: State s a -> s -> s
```

```
import Control.Monad.State

test :: State Int Int
test = do
  put 3
  modify (+1)
  get

main :: IO ()
main = print $ execState test 0
```

The state monad is often mistakenly described as being impure, but it is in fact entirely pure and the same effect could be achieved by explicitly passing state. A simple implementation of the State monad is only a few lines:

```haskell
newtype State s a = State { runState :: s -> (a,s) }

instance Monad (State s) where
  return a = State $ \s -> (a, s)

  State act >>= k = State $ \s ->
    let (a, s') = act s
    in runState (k a) s'

get :: State s s
get = State $ \s -> (s, s)

put :: s -> State s ()
put s = State $ \_ -> ((), s)

modify :: (s -> s) -> State s ()
modify f = get >>= \x -> put (f x)

evalState :: State s a -> s -> a
evalState act = fst . runState act

execState :: State s a -> s -> s
execState act = snd . runState act
```

Monad Tutorials

So many monad tutorials have been written that it begs the question, what makes monads so difficult when first learning Haskell. I suggest there are three aspects to why this is so:

1. *There are several levels on indirection with desugaring.*

A lot of Haskell that we write is radically rearranged and transformed into an entirely new form under the hood.

Most monad tutorials will not manually expand out the do-sugar. This leaves the beginner thinking that monads are a way of dropping into a pseudo-imperative language inside of code and further fuels that misconception that specific instances like IO are monads in their full generality.

```haskell
main = do
  x <- getLine
  putStrLn x
  return ()
```

Being able to manually desugar is crucial to understanding.

```
main =
  getLine >>= \x ->
    putStrLn x >>= \_ ->
      return ()
```

2. *Asymmetric binary infix operators for higher order functions are not common in other languages.*

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

On the left hand side of the operator we have an `m a` and on the right we have `a -> m b`. Although some languages do have infix operators that are themselves higher order functions, it is still a rather rare occurrence.

So with a function desugared, it can be confusing that `(>>=)` operator is in fact building up a much larger function by composing functions together.

```
main =
  getLine >>= \x ->
    putStrLn >>= \_ ->
      return ()
```

Written in prefix form, it becomes a little bit more digestible.

```
main =
  (>>=) getLine (\x ->
    (>>=) putStrLn (\_ ->
        return ()
    )
  )
```

Perhaps even removing the operator entirely might be more intuitive coming from other languages.

```
main = bind getLine (\x -> bind putStrLn (\_ -> return ()))
  where
    bind x y = x >>= y
```

3. *Ad-hoc polymorphism is not commonplace in other languages.*

Haskell's implementation of overloading can be unintuitive if one is not familiar with type inference. It is abstracted away from the user but the `(>>=)` or `bind` function is really a function of 3 arguments with the extra typeclass dictionary argument (`$dMonad`) implicitly threaded around.

```
main $dMonad = bind $dMonad getLine (\x -> bind $dMonad putStrLn (\_ -> return $dMonad
```

Except in the case where the parameter of the monad class is unified (through inference) with a concrete class instance, in which case the instance dictionary (`$dMonadIO`) is instead spliced throughout.

```
main :: IO ()
main = bind $dMonadIO getLine (\x -> bind $dMonadIO putStrLn (\_ -> return $dMonadIO (
```

Now, all of these transformations are trivial once we understand them, they're just typically not discussed. In my opinion the fundamental fallacy of monad tutorials is not that intuition for monads is hard to convey (nor are metaphors required!), but that novices often come to monads with an incomplete understanding of points (1), (2), and (3) and then trip on the simple fact that monads are the first example of a Haskell construct that is the confluence of all three.

See: Monad Tutorial Fallacy

Monad Transformers

mtl / transformers

So the descriptions of Monads in the previous chapter are a bit of a white lie. Modern Haskell monad libraries typically use a more general form of these written in terms of monad transformers which allow us to compose monads together to form composite monads. The monads mentioned previously are subsumed by the special case of the transformer form composed with the Identity monad.

	Monad Transformer Type		Transformed Type
Maybe	MaybeT	`Maybe a`	`m (Maybe a)`
Reader	ReaderT	`r -> a`	`r -> m a`
Writer	WriterT	`(a, w)`	`m (a, w)`
State	StateT	`s -> (a, s)`	`s -> m (a, s)`

```
type State  s = StateT  s Identity
type Writer w = WriterT w Identity
type Reader r = ReaderT r Identity

instance Monad m => MonadState s (StateT s m)
instance Monad m => MonadReader r (ReaderT r m)
instance (Monoid w, Monad m) => MonadWriter w (WriterT w m)
```

In terms of generality the mtl library is the most common general interface for these monads, which itself depends on the transformers library which generalizes the "basic" monads described above into transformers.

Transformers

At their core monad transformers allow us to nest monadic computations in a stack with an interface to exchange values between the levels, called `lift`.

```haskell
lift :: (Monad m, MonadTrans t) => m a -> t m a
liftIO :: MonadIO m => IO a -> m a
```

```haskell
class MonadTrans t where
    lift :: Monad m => m a -> t m a

class (Monad m) => MonadIO m where
    liftIO :: IO a -> m a

instance MonadIO IO where
    liftIO = id
```

Just as the base monad class has laws, monad transformers also have several laws:

Law #1

```haskell
lift . return = return
```

Law #2

```haskell
lift (m >>= f) = lift m >>= (lift . f)
```

Or equivalently:

Law #1

```haskell
  lift (return x)

= return x
```

Law #2

```haskell
  do x <- lift m
     lift (f x)

= lift $ do x <- m
            f x
```

It's useful to remember that transformers compose outside-in but are unrolled inside out.

Compose transformers outside-in.

```haskell
newtype Parser a = Parser { unParser :: WriterT [Int] (StateT String []) a }
```

```haskell
runParser p s = runStateT (runWriterT (unParser p)) s
```

Unroll transformers inside-out.

See: Monad Transformers: Step-By-Step

ReaderT

For example, there exist three possible forms of th Reader monad. The first is the Haskell 98 version that no longer exists but is useful for pedagogy. The other two are the *transformers* variant and the *mtl* variants.

Reader

```haskell
newtype Reader r a = Reader { runReader :: r -> a }

instance MonadReader r (Reader r) where
  ask       = Reader id
  local f m = Reader $ runReader m . f
```

ReaderT

```haskell
newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }

instance (Monad m) => Monad (ReaderT r m) where
  return a = ReaderT $ \_ -> return a
  m >>= k  = ReaderT $ \r -> do
      a <- runReaderT m r
      runReaderT (k a) r

instance MonadTrans (ReaderT r) where
    lift m = ReaderT $ \_ -> m
```

MonadReader

```haskell
class (Monad m) => MonadReader r m | m -> r where
  ask   :: m r
  local :: (r -> r) -> m a -> m a

instance (Monad m) => MonadReader r (ReaderT r m) where
  ask       = ReaderT return
  local f m = ReaderT $ \r -> runReaderT m (f r)
```

So hypothetically the three variants of ask would be:

```
ask :: Reader r a
ask :: Monad m => ReaderT r m r
ask :: MonadReader r m => m r
```

In practice only the last one is used in modern Haskell.

Basics

The most basic use requires us to use the T-variants of each of the monad transformers for the outer layers and to explicit `lift` and `return` values between each the layers. Monads have kind `(* -> *)` so monad transformers which take monads to monads have `((* -> *) -> * -> *)`:

```
Monad (m :: * -> *)
MonadTrans (t :: (* -> *) -> * -> *)
```

So for example if we wanted to form a composite computation using both the Reader and Maybe monads we can now put the Maybe inside of a `ReaderT` to form `ReaderT t Maybe a`.

```haskell
import Control.Monad.Reader

type Env = [(String, Int)]
type Eval a = ReaderT Env Maybe a

data Expr
  = Val Int
  | Add Expr Expr
  | Var String
  deriving (Show)

eval :: Expr -> Eval Int
eval ex = case ex of

  Val n -> return n

  Add x y -> do
    a <- eval x
    b <- eval y
    return (a+b)

  Var x -> do
    env <- ask
    val <- lift (lookup x env)
    return val

env :: Env
env = [("x", 2), ("y", 5)]

ex1 :: Eval Int
```

```
ex1 = eval (Add (Val 2) (Add (Val 1) (Var "x")))

example1, example2 :: Maybe Int
example1 = runReaderT ex1 env
example2 = runReaderT ex1 []
```

The fundamental limitation of this approach is that we find ourselves `lift.lift.lift` ing and `return.return.return` ing a lot.

Newtype Deriving

Newtypes let us reference a data type with a single constructor as a new distinct type, with no runtime overhead from boxing, unlike an algebraic datatype with single constructor. Newtype wrappers around strings and numeric types can often drastically reduce accidental errors. Using `XGeneralizedNewtypeDeriving` we can recover the functionality of instances of the underlying type.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype Velocity = Velocity { unVelocity :: Double }
  deriving (Eq, Ord)

v :: Velocity
v = Velocity 2.718

x :: Double
x = 6.636

-- Type error is caught at compile time even though they are the same value at runtime
err = v + x

newtype Quantity v a = Quantity a
  deriving (Eq, Ord, Num, Show)

data Haskeller
type Haskellers = Quantity Haskeller Int

a = Quantity 2 :: Haskellers
b = Quantity 6 :: Haskellers

totalHaskellers :: Haskellers
totalHaskellers = a + b
```

```
Couldn't match type `Double' with `Velocity'
Expected type: Velocity
  Actual type: Double
In the second argument of `(+)', namely `x'
In the expression: v + x
```

Using newtype deriving with the mtl library typeclasses we can produce flattened transformer types that don't require explicit lifting in the transform stack. For example, here is a little stack machine involving the Reader, Writer and State monads.

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

import Control.Monad.Reader
import Control.Monad.Writer
import Control.Monad.State

type Stack   = [Int]
type Output  = [Int]
type Program = [Instr]

type VM a = ReaderT Program (WriterT Output (State Stack)) a

newtype Comp a = Comp { unComp :: VM a }
  deriving (Monad, MonadReader Program, MonadWriter Output, MonadState Stack)

data Instr = Push Int | Pop | Puts

evalInstr :: Instr -> Comp ()
evalInstr instr = case instr of
  Pop     -> modify tail
  Push n -> modify (n:)
  Puts    -> do
    tos <- gets head
    tell [tos]

eval :: Comp ()
eval = do
  instr <- ask
  case instr of
    []      -> return ()
    (i:is) -> evalInstr i >> local (const is) eval

execVM :: Program -> Output
execVM = flip evalState [] . execWriterT . runReaderT (unComp eval)

program :: Program
program = [
    Push 42,
    Push 27,
    Puts,
    Pop,
    Puts,
    Pop
  ]

main :: IO ()
main = mapM_ print $ execVM program
```

Pattern matching on a newtype constructor compiles into nothing. For example the `extractB` function does not scrutinize the `MkB` constructor like the `extractA` does, because `MkB` does not exist at runtime, it is purely a compile-time construct.

```haskell
data A = MkA Int
newtype B = MkB Int

extractA :: A -> Int
extractA (MkA x) = x

extractB :: B -> Int
extractB (MkB x) = x
```

Efficiency

The second monad transformer law guarantees that sequencing consecutive lift operations is semantically equivalent to lifting the results into the outer monad.

```haskell
do x <- lift m   ==   lift $ do x <- m
   lift (f x)                   f x
```

Although they are guaranteed to yield the same result, the operation of lifting the results between the monad levels is not without cost and crops up frequently when working with the monad traversal and looping functions. For example, all three of the functions on the left below are less efficient than the right hand side which performs the bind in the base monad instead of lifting on each iteration.

```haskell
-- Less Efficient       More Efficient
forever (lift m)     == lift (forever m)
mapM_ (lift . f) xs  == lift (mapM_ f xs)
forM_ xs (lift . f)  == lift (forM_ xs f)
```

Monad Morphisms

```haskell
lift :: Monad m => m a -> t m a
```

```haskell
hoist  :: Monad m => (forall a. m a -> n a) -> t m b -> t n b
embed  :: Monad n => (forall a. m a -> t n a) -> t m b -> t n b
squash :: (Monad m, MMonad t) => t (t m) a -> t m a
```

TODO

See: mmorph

Language Extensions

It's important to distinguish between different categories of language extensions:

The inherent problem with classifying the extensions into the **General** and **Specialized** categories is that it's a subjective classification. Haskellers who do type astronautics will have a very different interpretation of Haskell than people who do database programming. As such this is a conservative assessment, as an arbitrary baseline let's consider `FlexibleInstances` and `OverloadedStrings` "everyday" while `GADTs` and `TypeFamilies` are "specialized".

Key

- *Benign* implies that importing the extension won't change the semantics of the module if not used.
- *Historical* implies that one shouldn't use this extension, it's in GHC purely for backwards compatibility. Sometimes these are dangerous to enable.

	Benign	Historical	Extends Syntax	Use	Use	GHC Reference
AllowAmbiguousTypes				Specialized	Typelevel Programming	Ref
Arrows			✓	Specialized	Syntax Extension	Ref
AutoDeriveTypeable				Specialized	Metaprogramming	Ref
BangPatterns	✓		✓	General	Strictness Annotation	Ref
CApiFFI				Specialized	FFI	Ref
ConstrainedClassMethods				Specialized	Typelevel Programming	Ref
ConstraintKinds				Specialized	Typelevel Programming	Ref
CPP	✓		✓	General	Preprocessor	Ref
DataKinds				Specialized	Typelevel Programming	Ref
DatatypeContexts		✓	✓	Deprecated	Deprecated	Ref
DefaultSignatures	✓			Specialized	Generic Programming	Ref
DeriveDataTypeable	✓			General	Generic Programming	Ref
DeriveFoldable	✓			General	Generic Programming	Ref
DeriveFunctor	✓			General	Generic Programming	Ref
DeriveGeneric	✓			General	Generic Programming	Ref
DeriveTraversable	✓			General	Generic Programming	Ref
DisambiguateRecordFields	✓		✓	Specialized	Syntax Extension	Ref
DoRec		✓	✓	Specialized	Syntax Extension	Ref
EmptyCase				Specialized	Syntax Extension	Ref
EmptyDataDecls	✓			General	Syntax Extension	Ref
ExistentialQuantification				Specialized	Typelevel Programming	Ref
ExplicitForAll			✓	Specialized	Typelevel Programming	Ref
ExplicitNamespaces	✓		✓	Specialized	Syntax Disambiguation	Ref
ExtendedDefaultRules	✓			Specialized	Generic Programming	Ref
FlexibleContexts				General	Typeclass	Ref

Extension					
FlexibleInstances			General	Typeclass Extension	Ref
ForeignFunctionInterface		✓	General	FFI	Ref
FunctionalDependencies			General	Typeclass Extension	Ref
GADTs			General	Typelevel Programming	Ref
GADTSyntax		✓	General	Syntax Extension	Ref
GeneralizedNewtypeDeriving			General	Typeclass Extension	Ref
GHCForeignImportPrim			Specialized	FFI	Ref
ImplicitParams			Specialized	Typelevel Programming	Ref
ImpredicativeTypes			Specialized	Typelevel Programming	Ref
IncoherentInstances			Specialized	Typelevel Programming	Ref
InstanceSigs			Specialized	Typelevel Programming	Ref
InterruptibleFFI			Specialized	FFI	Ref
KindSignatures			Specialized	Typelevel Programming	Ref
LambdaCase	✓	✓	General	Syntax Extension	Ref
LiberalTypeSynonyms			Specialized	Typeclass Extension	Ref
MagicHash			Specialized	GHC Internals	Ref
MonadComprehensions		✓	Specialized	Syntax Extension	Ref
MonoPatBinds			Specialized	Type Disambiguation	Ref
MultiParamTypeClasses	✓		General	Typeclass Extension	Ref
MultiWayIf		✓	Specialized	Syntax Extension	Ref
NamedFieldPuns		✓	Specialized	Syntax Extension	Ref
NegativeLiterals			General	Type Disambiguation	Ref
NoImplicitPrelude			Specialized	Import Disambiguation	Ref
NoMonoLocalBinds			General	Type Disambiguation	Ref
NoMonomorphismRestriction			General	Type Disambiguation	Ref
NPlusKPatterns	✓	✓	Deprecated	Deprecated	Ref
NullaryTypeClasses			Specialized	Typeclass Extension	Ref
NumDecimals			General	Type Disambiguation	Ref
OverlappingInstances			Specialized	Typeclass Extension	Ref
OverloadedLists		✓	General	Syntax Extension	Ref
OverloadedStrings			General	Syntax Extension	Ref
PackageImports		✓	General	Import Disambiguation	Ref
ParallelArrays			Specialized	Data Parallel Haskell	Ref
ParallelListComp		✓	General	Syntax Extension	Ref
PatternGuards		✓	General	Syntax Extension	Ref
PatternSynonyms			General	Syntax Extension	Ref

Extension						
	✓		✓			
PolyKinds				Specialized	Typelevel Programming	Ref
PolymorphicComponents		✓		Specialized	Deprecated	Ref
PostfixOperators	✓		✓	Specialized	Syntax Extension	Ref
QuasiQuotes				Specialized	Metaprogramming	Ref
Rank2Types		✓		Specialized	Historical Artifact	Ref
RankNTypes				Specialized	Typelevel Programming	Ref
RebindableSyntax			✓	Specialized	Metaprogramming	Ref
RecordWildCards	✓		✓	General	Syntax Extension	Ref
RecursiveDo				Specialized	Syntax Extension	Ref
RelaxedPolyRec				Specialized	Type Disambiguation	Ref
RoleAnnotations				Specialized	Type Disambiguation	Ref
Safe				Specialized	Security Auditing	Ref
SafeImports				Specialized	Security Auditing	Ref
ScopedTypeVariables				Specialized	Typelevel Programming	Ref
StandaloneDeriving	✓		✓	General	Typeclass Extension	Ref
TemplateHaskell	✓		✓	Specialized	Metaprogramming	Ref
TraditionalRecordSyntax		✓	✓	Specialized	Historical Artifact	Ref
TransformListComp			✓	Specialized	Syntax Extension	Ref
Trustworthy				Specialized	Security Auditing	Ref
TupleSections	✓			General	Syntax Extension	Ref
TypeFamilies				Specialized	Typelevel Programming	Ref
TypeHoles	✓			General	Interactive Typing	Ref
TypeOperators				Specialized	Typelevel Programming	Ref
TypeSynonymInstances	✓			General	Typeclass Extension	Ref
UnboxedTuples				Specialized	FFI	Ref
UndecidableInstances				Specialized	Typelevel Programming	Ref
UnicodeSyntax			✓	Specialized	Syntax Extension	Ref
UnliftedFFITypes				Specialized	FFI	Ref
Unsafe				Specialized	Security Auditing	Ref
ViewPatterns	✓		✓	General	Syntax Extension	Ref

See: GHC Extension Reference

The Benign

It's not obvious which extensions are the most common but it's fairly safe to say that these extensions are benign and are safely used extensively:

- NoMonomorphismRestriction
- FlexibleContexts
- FlexibleInstances
- GeneralizedNewtypeDeriving
- GADTs
- FunctionalDependencies
- OverloadedStrings
- TypeSynonymInstances

- BangPatterns
- DeriveGeneric
- DeriveDataTypeable
- ScopedTypeVariables

The Dangerous

GHC's typechecker sometimes just casually tells us to enable language extensions when it can't solve certain problems. These include:

- DatatypeContexts
- OverlappingInstances
- IncoherentInstances
- ImpredicativeTypes

These almost always indicate a design flaw and shouldn't be turned on to remedy the error at hand, as much as GHC might suggest otherwise!

Inference

Inference in Haskell is generally quite accurate, although there are several boundary cases that tend to cause problems. Consider the two functions

Mutually Recursive Binding Groups

```
f x = const x g
g y = f 'A'
```

The inferred type signatures are correct in their usage, but don't represent the most general signatures. When GHC analyzes the module it analyzes the dependencies of expressions on each other, groups them together, and applies substitutions from unification across mutually defined groups. As such the inferred types may not be the most general types possible, and an explicit signature may be desired.

```
-- Inferred types
f :: Char -> Char
g :: t -> Char

-- Most general types
f :: a -> a
g :: a -> Char
```

Polymorphic recursion

```
data Tree a = Leaf | Bin a (Tree (a, a))

size Leaf = 0
size (Bin _ t) = 1 + 2 * size t
```

The problem with this expression is because the inferred type variable (a) in (size) spans two possible types ((a) and (a, a)), the recursion is polymorphic. These two types won't pass the occurs-

check of the typechecker and it yields an incorrect inferred type.

```
Occurs check: cannot construct the infinite type: t0 = (t0, t0)
Expected type: Tree t0
  Actual type: Tree (t0, t0)
In the first argument of 'size', namely 't'
In the second argument of '(*)', namely 'size t'
In the second argument of '(+)', namely '2 * size t'
```

Simply adding an explicit type signature corrects this. Type inference using polymorphic recursion is undecidable in the general case.

```
size :: Tree a -> Int
size Leaf = 0
size (Bin _ t) = 1 + 2 * size t
```

See: Static Semantics of Function and Pattern Bindings

Monomorphism Restriction

The most common edge case of the inference is known as the dreaded *monomorphic restriction*.

When the toplevel declarations of a module are generalized the monomorphism restricts that toplevel values (i.e. expressions not under a lambda) whose type contains the subclass of the `Num` type from the Prelude are not generalized and instead are instantiated with a monotype tried sequentially from the list specified by the `default` which is normally `Integer`, then `Double`.

```
-- Double is inferred by type inferencer.
example1 :: Double
example1 = 3.14

-- In the presense of a lambda, a different type is inferred!
example2 :: Fractional a => t -> a
example2 _ = 3.14

default (Integer, Double)
```

As of GHC 7.8, the monomorphism restriction is switched off by default in GHCi.

```
λ: set +t

λ: 3
3
it :: Num a => a

λ: default (Double)
```

```
λ: 3
3.0
it :: Num a => a
```

Safe Haskell

As everyone eventually finds out there are several functions within the implementation of GHC (not the Haskell language) that can be used to subvert the type-system, they are marked with the prefix `unsafe`. These functions exist only for when one can manually prove the soundness of an expression but can't express this property in the type-system. Using these functions without fulfilling the proof obligations will cause all measure of undefined behavior with unimaginable pain and suffering, and are **strongly discouraged**. When initially starting out with Haskell there are no legitimate reason to use these functions at all, period.

```
unsafeCoerce :: a -> b
unsafePerformIO :: IO a -> a
```

The Safe Haskell language extensions allow us to restrict the use of unsafe language features using `XSafe` which restricts the import of modules which are themselves marked as Safe. It also forbids the use of certain language extensions (`-XTemplateHaskell`) which can be used to produce unsafe code. The primary use case of these extensions is security auditing.

```
{-# LANGUAGE Safe #-}
{-# LANGUAGE Trustworthy #-}
```

```
{-# LANGUAGE Safe #-}

import Unsafe.Coerce
import System.IO.Unsafe

bad1 :: String
bad1 = unsafePerformIO getLine

bad2 :: a
bad2 = unsafeCoerce 3.14 ()
```

```
Unsafe.Coerce: Can't be safely imported!
The module itself isn't safe.
```

See: <u>Safe Haskell</u>

Pattern Guards

```
{-# LANGUAGE PatternGuards #-}
```

```
combine env x y
   | Just a <- lookup x env
   , Just b <- lookup y env
   = Just $ a + b

   | otherwise = Nothing
```

View Patterns

```
{-# LANGUAGE ViewPatterns #-}
{-# LANGUAGE NoMonomorphismRestriction #-}

import Safe

lookupDefault :: Eq a => a -> b -> [(a,b)] -> b
lookupDefault k _ (lookup k -> Just s) = s
lookupDefault _ d _ = d

headTup :: (a, [t]) -> [t]
headTup (headMay . snd -> Just n) = [n]
headTup _ = []

headNil :: [a] -> [a]
headNil (headMay -> Just x) = [x]
headNil _ = []
```

Misc Syntax Extensions

Tuple Sections

```
{-# LANGUAGE TupleSections #-}

first :: a -> (a, Bool)
first = (,True)

second :: a -> (Bool, a)
second = (True,)
```

Multi-way if-expressions

```
{-# LANGUAGE MultiWayIf #-}

operation x =
  if | x > 100  = 3
     | x > 10   = 2
```

```
    | x > 1     = 1
    | otherwise = 0
```

Lambda Case

```
{-# LANGUAGE LambdaCase #-}

data Exp a
  = Lam a (Exp a)
  | Var a
  | App (Exp a) (Exp a)

example :: Exp a -> a
example = \case
  Lam a b -> a
  Var a   -> a
  App a b -> example a
```

Package Imports

```
import qualified "mtl" Control.Monad.Error as Error
import qualified "mtl" Control.Monad.State as State
import qualified "mtl" Control.Monad.Reader as Reader
```

Record WildCards

Record wild cards allow us to expand out the names of a record as variables scoped as the labels of the record implicitly.

```
{-# LANGUAGE RecordWildCards #-}

data T = T { a :: Int , b :: Int }

f :: T -> Int
f (T {..} ) = a + b
```

Pattern Synonyms

Suppose we were writing a typechecker, it would be very common to include a distinct $\boxed{TArr}$ term to ease the telescoping of function signatures, this is what GHC does in its Core language. Even though technically it could be written in terms of more basic application of the $\boxed{(\text{->})}$ constructor.

```
data Type
  = TVar TVar
  | TCon TyCon
```

```
  | TApp Type Type
  | TArr Type Type
  deriving (Show, Eq, Ord)
```

With pattern synonyms we can eliminate the extraneous constructor without losing the convenience of
pattern matching on arrow types.

```
{-# LANGUAGE PatternSynonyms #-}

pattern TArr t1 t2 = TApp (TApp (TCon "(->)") t1) t2
```

So now we can write an eliminator and constructor for arrow type very naturally.

```
{-# LANGUAGE PatternSynonyms #-}

import Data.List (foldl1')

type Name  = String
type TVar  = String
type TyCon = String

data Type
  = TVar TVar
  | TCon TyCon
  | TApp Type Type
  deriving (Show, Eq, Ord)


pattern TArr t1 t2 = TApp (TApp (TCon "(->)") t1) t2

tapp :: TyCon -> [Type] -> Type
tapp tcon args = foldl TApp (TCon tcon) args

arr :: [Type] -> Type
arr ts = foldl1' (\t1 t2 -> tapp "(->)" [t1, t2]) ts

elimTArr :: Type -> [Type]
elimTArr (TArr (TArr t1 t2) t3) = t1 : t2 : elimTArr t3
elimTArr (TArr t1 t2) = t1 : elimTArr t2
elimTArr t = [t]

-- (->) a ((->) b a)
-- a -> b -> a
to :: Type
to = arr [TVar "a", TVar "b", TVar "a"]

from :: [Type]
from = elimTArr to
```

Laziness

Again, a subject on which *much* ink has been spilled. There is an ongoing discussion in the land of Haskell about the compromises between lazy and strict evaluation, and there are nuanced arguments for having either paradigm be the default. Haskell takes a hybrid approach and allows strict evaluation when needed and uses laziness by default. Needless to say, we can always find examples where strict evaluation exhibits worse behavior than lazy evaluation and vice versa.

The primary advantage of lazy evaluation in the large is that algorithms that operate over both unbounded and bounded data structures can inhabit the same type signatures and be composed without additional need to restructure their logic or force intermediate computations. Languages that attempt to bolt laziness on to a strict evaluation model often bifurcate classes of algorithms into ones that are hand-adjusted to consume unbounded structures and those which operate over bounded structures. In strict languages mixing and matching between lazy vs strict processing often necessitates manifesting large intermediate structures in memory when such composition would "just work" in a lazy language.

By virtue of Haskell being the only language to actually explore this point in the design space to the point of being industrial strength; knowledge about lazy evaluation is not widely absorbed into the collective programmer consciousness and can often be non-intuitive to the novice. This does reflect on the model itself, merely on the need for more instruction material and research on optimizing lazy compilers.

The paradox of Haskell is that it explores so many definably unique ideas (laziness, purity, typeclasses) that it becomes difficult to separate out the discussion of any one from the gestalt of the whole implementation.

See:

- Oh My Laziness!
- Reasoning about Laziness
- Lazy Evaluation of Haskell
- More Points For Lazy Evaluation
- How Lazy Evaluation Works in Haskell

Strictness

There are several evaluation models for the lambda calculus:

- Strict - Evaluation is said to be strict if all arguments are evaluated before the body of a function.
- Non-strict - Evaluation is non-strict if the arguments are not necessarily evaluated before entering the body of a function.

These ideas give rise to several models, Haskell itself use the *call-by-need* model.

Model	Strictness	Description
Call-by-value	Strict	arguments evaluated before function entered
Call-by-name	Non-strict	arguments passed unevaluated
Call-by-need	Non-strict	arguments passed unevaluated but an expression is only evaluated once (sharing)

Seq and WHNF

A term is said to be in *weak head normal-form* if the outermost constructor or lambda cannot be reduced further. A term is said to be in *normal form* if it is fully evaluated and all sub-expressions and thunks contained within are evaluated.

```
-- In Normal Form
```

```
42
(2, "foo")
\x -> x + 1

-- Not in Normal Form
1 + 2
(\x -> x + 1) 2
"foo" ++ "bar"
(1 + 1, "foo")

-- In Weak Head Normal Form
(1 + 1, "foo")
\x -> 2 + 2
'f' : ("oo" ++ "bar")

-- Not In Weak Head Normal Form
1 + 1
(\x -> x + 1) 2
"foo" ++ "bar"
```

In Haskell normal evaluation only occurs at the outer constructor of case-statements in Core. If we pattern match on a list we don't implicitly force all values in the list. An element in a data structure is only evaluated up to the most outer constructor. For example, to evaluate the length of a list we need only scrutinize the outer Cons constructors without regard for their inner values.

```
λ: length [undefined, 1]
2

λ: head [undefined, 1]
Prelude.undefined

λ: snd (undefined, 1)
1

λ: fst (undefined, 1)
Prelude.undefined
```

For example, in a lazy language the following program terminates even though it contains diverging terms.

```
ignore :: a -> Int
ignore x = 0

loop :: a
loop = loop

main :: IO ()
main = print $ ignore loop
```

In a strict language like OCaml (ignoring its suspensions for the moment), the same program diverges.

```
let ignore x = 0;;
let rec loop a = loop a;;

print_int (ignore (loop ()));
```

In Haskell a *thunk* is created to stand for an unevaluated computation. Evaluation of a thunk is called *forcing* the thunk. The result is an *update*, a referentially transparent effect, which replaces the memory representation of the thunk with the computed value. The fundamental idea is that a thunk is only updated once (although it may be forced simultaneously in a multi-threaded environment) and its resulting value is shared when referenced subsequently.

The command `:sprint` can be used to introspect the state of unevaluated thunks inside an expression without forcing evaluation. For instance:

```
λ: let a = [1..] :: [Integer]
λ: let b = map (+ 1) a

λ: :sprint a
a = _
λ: :sprint b
b = _
λ: a !! 4
5
λ: :sprint a
a = 1 : 2 : 3 : 4 : 5 : _
λ: b !! 10
12
λ: :sprint a
a = 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : 10 : 11 : _
λ: :sprint b
b = _ : _ : _ : _ : _ : _ : _ : _ : _ : _ : 12 : _
```

While a thunk is being computed its memory representation is replaced with a special form known as *blackhole* which indicates that computation is ongoing and allows for a short circuit for when a computation might depend on itself to complete. The implementation of this is some of the more subtle details of the GHC runtime.

The `seq` function introduces an artificial dependence on the evaluation of order of two terms by requiring that the first argument be evaluated to WHNF before the evaluation of the second. The implementation of the `seq` function is an implementation detail of GHC.

```
seq :: a -> b -> b

⊥ `seq` a = ⊥
a `seq` b = b
```

The infamous `foldl` is well-known to leak space when used carelessly and without several compiler

optimizations applied. The strict `foldl'` variant uses seq to overcome this.

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

```haskell
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' _ z [] = z
foldl' f z (x:xs) = let z' = f z x in z' `seq` foldl' f z' xs
```

In practice, a combination between the strictness analyzer and the inliner on `-O2` will ensure that the strict variant of `foldl` is used whenever the function is inlinable at call site so manually using `foldl'` is most often not required.

Of important note is that GHCi runs without any optimizations applied so the same program that performs poorly in GHCi may not have the same performance characteristics when compiled with GHC.

Strictness Annotations

The extension `BangPatterns` allows an alternative syntax to force arguments to functions to be wrapped in seq. A bang operator on an arguments forces its evaluation to weak head normal form before performing the pattern match. This can be used to keep specific arguments evaluated throughout recursion instead of creating a giant chain of thunks.

```haskell
{-# LANGUAGE BangPatterns #-}

sum :: Num a => [a] -> a
sum = go 0
  where
    go !acc (x:xs) = go (acc + x) (go xs)
    go  acc []     = acc
```

This is desugared into code effectively equivalent to the following:

```haskell
sum :: Num a => [a] -> a
sum = go 0
  where
    go acc _ | acc `seq` False = undefined
    go acc (x:xs)              = go (acc + x) (go xs)
    go acc []                  = acc
```

Function application to seq'd arguments is common enough that it has a special operator.

```haskell
($!) :: (a -> b) -> a -> b
f $! x  = let !vx = x in f vx
```

Deepseq

There are often times when for performance reasons we need to deeply evaluate a data structure to normal form leaving no terms unevaluated. The `deepseq` library performs this task.

The typeclass `NFData` (Normal Form Data) allows us to seq all elements of a structure across any subtypes which themselves implement NFData.

```haskell
class NFData a where
  rnf :: a -> ()
  rnf a = a `seq` ()

deepseq :: NFData a => a -> b -> a
($!!) :: (NFData a) => (a -> b) -> a -> b
```

```haskell
instance NFData Int
instance NFData (a -> b)

instance NFData a => NFData (Maybe a) where
    rnf Nothing  = ()
    rnf (Just x) = rnf x

instance NFData a => NFData [a] where
    rnf [] = ()
    rnf (x:xs) = rnf x `seq` rnf xs
```

```haskell
[1, undefined] `seq` ()
-- ()

[1, undefined] `deepseq` ()
-- Prelude.undefined
```

To force a data structure itself to be fully evaluated we share the same argument in both positions of deepseq.

```haskell
force :: NFData a => a
force x = x `deepseq` x
```

Irrefutable Patterns

A lazy pattern doesn't require a match on the outer constructor, instead it lazily calls the accessors of the values failing at each call-site instead at the outer pattern match in the presence of a bottom.

```haskell
f :: (a, b) -> Int
```

```
 f (a,b) = const 1 a

 g :: (a, b) -> Int
 g ~(a,b) = const 1 a

 -- λ: f undefined
 -- *** Exception: Prelude.undefined
 -- λ: g undefined
 -- 1

 j :: Maybe t -> t
 j ~(Just x) = x

 k :: Maybe t -> t
 k (Just x) = x

 -- λ: j Nothing
 -- *** Exception: src/05-laziness/lazy_patterns.hs:15:1-15: Irrefutable pattern failed
 --
 -- λ: k Nothing
 -- *** Exception: src/05-laziness/lazy_patterns.hs:18:1-14: Non-exhaustive patterns in
```

Moral Correctness

The caveat with lazy evaluation is that it implies inductive reasoning about functions, because we must always take into account the fact that a function may contain bottoms. And as such claims about inductive proofs of functions have to be couched in an implied set of qualifiers "up to the fast and loose reasoning" assuming the non-existence of bottoms.

In the "Fast and Loose Reasoning is Morally Correct" paper John Hughes et al. showed that if two terms have the same semantics in the total language, then they have related semantics in the partial language and gave a prescription by which we can translate our knowledge between the two domains given a specific set of finely stated conditions under which proofs about lazy languages are indeed rigorous and sound.

- Fast and Loose Reasoning is Morally Correct

Prelude

What to Avoid?

Haskell being a 25 year old language has witnessed several revolutions in the way we structure and compose functional programs. Yet as a result several portions of the Prelude still reflect old schools of thought that simply can't be removed without breaking significant parts of the ecosystem.

Currently it really only exists in folklore which parts to use and which not to use, although this is a topic that almost all introductory books don't mention and instead make extensive use of the Prelude for simplicity's sake.

The short version of the advice on the Prelude is:

- Use `fmap` instead of `map`.
- Use Foldable and Traversable instead of the Control.Monad, and Data.List versions of traversals.
- Avoid partial functions like `head` and `read` or use their total variants.
- Avoid asynchronous exceptions.

- Avoid boolean blind functions.

The instances of Foldable for the list type often conflict with the monomorphic versions in the Prelude which are left in for historical reasons. So often times it is desirable to explicitly mask these functions from implicit import and force the use of Foldable and Traversable instead:

```
import  Data.List hiding (
    all , and , any , concat , concatMap find , foldl ,
    foldl' , foldl1 , foldr , foldr1 , mapAccumL ,
    mapAccumR , maximum , maximumBy , minimum ,
    minimumBy , notElem , or , product , sum )

import Control.Monad hiding (
    forM , forM_ , mapM , mapM_ , msum , sequence , sequence_ )
```

Of course often times one wishes only to use the Prelude explicitly and one can explicitly import it qualified and use the pieces as desired without the implicit import of the whole namespace.

```
import qualified Prelude as P
```

This does however bring in several typeclass instances and classes regardless of whether it is explicitly or implicitly imported. If one really desires to use nothing from the Prelude then the option exists to exclude the entire prelude (except for the wired-in class instances) with the `-XNoImplicitPrelude` pragma.

```
{-# LANGUAGE NoImplicitPrelude #-}
```

The Prelude itself is entirely replicable as well presuming that an entire project is compiled without the implicit Prelude. Several packages have arisen that supply much of the same functionality in a way that appeals to more modern design principles.

- base-prelude
- basic-prelude
- classy-prelude

Partial Functions

A *partial function* is a function which doesn't terminate and yield a value for all given inputs. Conversely a *total function* terminates and is always defined for all inputs. As mentioned previously, certain historical parts of the Prelude are full of partial functions.

The difference between partial and total functions is the compiler can't reason about the runtime safety of partial functions purely from the information specified in the language and as such the proof of safety is left to the user to guarantee. They are safe to use in the case where the user can guarantee that invalid inputs cannot occur, but like any unchecked property its safety or not-safety is going to depend on the diligence of the programmer. This very much goes against the overall philosophy of Haskell and as such they are discouraged when not necessary.

```
head :: [a] -> a
read :: Read a => String -> a
```

```
(!!) :: [a] -> Int -> a
```

Safe

The Prelude has total variants of the historical partial functions (i.e. `Text.Read.readMaybe`)in some cases, but often these are found in the various utility libraries like `safe` .

The total versions provided fall into three cases:

- `May` - return Nothing when the function is not defined for the inputs
- `Def` - provide a default value when the function is not defined for the inputs
- `Note` - call `error` with a custom error message when the function is not defined for the inputs. This is not safe, but slightly easier to debug!

```
-- Total
headMay :: [a] -> Maybe a
readMay :: Read a => String -> Maybe a
atMay :: [a] -> Int -> Maybe a

-- Total
headDef :: a -> [a] -> a
readDef :: Read a => a -> String -> a
atDef   :: a -> [a] -> Int -> a

-- Partial
headNote :: String -> [a] -> a
readNote :: Read a => String -> String -> a
atNote   :: String -> [a] -> Int -> a
```

Boolean Blindness

```
data Bool = True | False

isJust :: Maybe a -> Bool
isJust (Just x) = True
isJust Nothing  = False
```

The problem with the boolean type is that there is effectively no difference between True and False at the type level. A proposition taking a value to a Bool takes any information given and destroys it. To reason about the behavior we have to trace the provenance of the proposition we're getting the boolean answer from, and this introduces a whole slew of possibilities for misinterpretation. In the worst case, the only way to reason about safe and unsafe use of a function is by trusting that a predicate's lexical name reflects its provenance!

For instance, testing some proposition over a Bool value representing whether the branch can perform the computation safely in the presence of a null is subject to accidental interchange. Consider that in a language like C or Python testing whether a value is null is indistinguishable to the language from testing whether the value is *not null*. Which of these programs encodes safe usage and which segfaults?

```python
# This one?
if p(x):
    # use x
elif not p(x):
    # don't use x

# Or this one?
if p(x):
    # don't use x
elif not p(x):
    # use x
```

For inspection we can't tell without knowing how p is defined, the compiler can't distinguish the two either and thus the language won't save us if we happen to mix them up. Instead of making invalid states *unrepresentable* we've made the invalid state *indistinguishable* from the valid one!

The more desirable practice is to match on terms which explicitly witness the proposition as a type (often in a sum type) and won't typecheck otherwise.

```haskell
case x of
  Just a  -> use x
  Nothing -> don't use x

-- not ideal
case p x of
  True  -> use x
  False -> don't use x

-- not ideal
if p x
  then use x
  else don't use x
```

To be fair though, many popular languages completely lack the notion of sum types (the source of many woes in my opinion) and only have product types, so this type of reasoning sometimes has no direct equivalence for those not familiar with ML family languages.

In Haskell, the Prelude provides functions like `isJust` and `fromJust` both of which can be used to subvert this kind of reasoning and make it easy to introduce bugs and should often be avoided.

Foldable / Traversable

If coming from an imperative background retraining one's self to think about iteration over lists in terms of maps, folds, and scans can be challenging.

```haskell
Prelude.foldl :: (a -> b -> a) -> a -> [b] -> a
Prelude.foldr :: (a -> b -> b) -> b -> [a] -> b

-- pseudocode
foldr f z [a...] = f a (f b ( ... (f y z) ... ))
foldl f z [a...] = f ... (f (f z a) b) ... y
```

For a concrete consider the simple arithmetic sequence over the binary operator `(+)`:

```
-- foldr (+) 1 [2..]
(1 + (2 + (3 + (4 + ...))))
```

```
-- foldl (+) 1 [2..]
((((1 + 2) + 3) + 4) + ...)
```

Foldable and Traversable are the general interface for all traversals and folds of any data structure which is parameterized over its element type (List, Map, Set, Maybe, ...). These two classes are used everywhere in modern Haskell and are extremely important.

A foldable instance allows us to apply functions to data types of monoidal values that collapse the structure using some logic over `mappend`.

A traversable instance allows us to apply functions to data types that walk the structure left-to-right within an applicative context.

```haskell
class (Functor f, Foldable f) => Traversable f where
  traverse :: Applicative g => f (g a) -> g (f a)

class Foldable f where
  foldMap :: Monoid m => (a -> m) -> f a -> m
```

The `foldMap` function is extremely general and non-intuitively many of the monomorphic list folds can themselves be written in terms of this single polymorphic function.

`foldMap` takes a function of values to a monoidal quantity, a functor over the values and collapses the functor into the monoid. For instance for the trivial Sum monoid:

```
λ: foldMap Sum [1..10]
Sum {getSum = 55}
```

The full Foldable class (with all default implementations) contains a variety of derived functions which themselves can be written in terms of `foldMap` and `Endo`.

```haskell
newtype Endo a = Endo {appEndo :: a -> a}

instance Monoid (Endo a) where
        mempty = Endo id
        Endo f `mappend` Endo g = Endo (f . g)


class Foldable t where
```

```
fold    :: Monoid m => t m -> m
foldMap :: Monoid m => (a -> m) -> t a -> m

foldr   :: (a -> b -> b) -> b -> t a -> b
foldr'  :: (a -> b -> b) -> b -> t a -> b

foldl   :: (b -> a -> b) -> b -> t a -> b
foldl'  :: (b -> a -> b) -> b -> t a -> b

foldr1  :: (a -> a -> a) -> t a -> a
foldl1  :: (a -> a -> a) -> t a -> a
```

For example:

```
foldr :: (a -> b -> b) -> b -> t a -> b
foldr f z t = appEndo (foldMap (Endo . f) t) z
```

Most of the operations over lists can be generalized in terms of combinations of Foldable and Traversable to derive more general functions that work over all data structures implementing Foldable.

```
Data.Foldable.elem    :: (Eq a, Foldable t) => a -> t a -> Bool
Data.Foldable.sum     :: (Num a, Foldable t) => t a -> a
Data.Foldable.minimum :: (Ord a, Foldable t) => t a -> a
Data.Traversable.mapM :: (Monad m, Traversable t) => (a -> m b) -> t a -> m (t b)
```

Unfortunately for historical reasons the names exported by foldable quite often conflict with ones defined in the Prelude, either import them qualified or just disable the Prelude. The operations in the Foldable all specialize to the same and behave the same as the ones in Prelude for List types.

```
import Data.Monoid
import Data.Foldable
import Data.Traversable

import Control.Applicative
import Control.Monad.Identity (runIdentity)
import Prelude hiding (mapM_, foldr)

-- Rose Tree
data Tree a = Node a [Tree a] deriving (Show)

instance Functor Tree where
  fmap f (Node x ts) = Node (f x) (fmap (fmap f) ts)

instance Traversable Tree where
  traverse f (Node x ts) = Node <$> f x <*> traverse (traverse f) ts

instance Foldable Tree where
  foldMap f (Node x ts) = f x `mappend` foldMap (foldMap f) ts
```

```
tree :: Tree Integer
tree = Node 1 [Node 1 [], Node 2 [] ,Node 3 []]


example1 :: IO ()
example1 = mapM_ print tree

example2 :: Integer
example2 = foldr (+) 0 tree

example3 :: Maybe (Tree Integer)
example3 = traverse (\x -> if x > 2 then Just x else Nothing) tree

example4 :: Tree Integer
example4 = runIdentity $ traverse (\x -> pure (x+1)) tree
```

The instances we defined above can also be automatically derived by GHC using several language extensions. The automatic instances are identical to the hand-written versions above.

```
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE DeriveFoldable #-}
{-# LANGUAGE DeriveTraversable #-}

data Tree a = Node a [Tree a]
  deriving (Show, Functor, Foldable, Traversable)
```

See: Typeclassopedia

Corecursion

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

A recursive function consumes data and eventually terminates, a corecursive function generates data and **coterminates**. A corecursive function is said to be *productive* if it can always evaluate more of the resulting value in bounded time.

```
import Data.List

f :: Int -> Maybe (Int, Int)
f 0 = Nothing
f x = Just (x, x-1)

rev :: [Int]
rev = unfoldr f 10
```

```haskell
fibs :: [Int]
fibs = unfoldr (\(a,b) -> Just (a,(b,a+b))) (0,1)
```

Split

The split package provides a variety of missing functions for splitting list and string types.

```haskell
import Data.List.Split

example1 :: [String]
example1 = splitOn "." "foo.bar.baz"
-- ["foo","bar","baz"]

example2 :: [String]
example2 = chunksOf 10 "To be or not to be that is the question."
-- ["To be or n","ot to be t","hat is the"," question."]
```

Monad-loops

The monad-loops package provides a variety of missing functions for control logic in monadic contexts.

```haskell
whileM :: Monad m => m Bool -> m a -> m [a]
untilM :: Monad m => m a -> m Bool -> m [a]
iterateUntilM :: Monad m => (a -> Bool) -> (a -> m a) -> a -> m a
whileJust :: Monad m => m (Maybe a) -> (a -> m b) -> m [b]
```

Text / ByteString

The default Haskell string type is the rather naive linked list of characters, that while perfectly fine for small identifiers is not well-suited for bulk processing.

```haskell
type String = [Char]
```

For more performance sensitive cases there are two libraries for processing textual data: `text` and `bytestring`. With the `-XOverloadedStrings` extension string literals can be overloaded without the need for explicit packing and can be written as string literals in the Haskell source and overloaded via a typeclass `IsString`.

```haskell
class IsString a where
  fromString :: String -> a
```

For instance:

```
λ: :type "foo"
"foo" :: [Char]

λ: :set -XOverloadedStrings

λ: :type "foo"
"foo" :: IsString a => a
```

Text

A `Text` type is a packed blob of Unicode characters.

```
pack :: String -> Text
unpack :: Text -> String
```

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

-- From pack
myTStr1 :: T.Text
myTStr1 = T.pack ("foo" :: String)

-- From overloaded string literal.
myTStr2 :: T.Text
myTStr2 = "bar"
```

See: Text

Text.Builder

```
toLazyText :: Builder -> Data.Text.Lazy.Internal.Text
fromLazyText :: Data.Text.Lazy.Internal.Text -> Builder
```

The Text.Builder allows the efficient monoidal construction of lazy Text types without having to go through inefficient forms like String or List types as intermediates.

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Monoid (mconcat, (<>))

import Data.Text.Lazy.Builder (Builder, toLazyText)
import Data.Text.Lazy.Builder.Int (decimal)
```

```
import qualified Data.Text.Lazy.IO as L

beer :: Int -> Builder
beer n = decimal n <> " bottles of beer on the wall.\n"

wall :: Builder
wall = mconcat $ fmap beer [1..1000]

main :: IO ()
main = L.putStrLn $ toLazyText wall
```

ByteString

ByteStrings are arrays of unboxed characters with either strict or lazy evaluation.

```
pack :: String -> ByteString
unpack :: ByteString -> String
```

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.ByteString as S
import qualified Data.ByteString.Char8 as S8

-- From pack
bstr1 :: S.ByteString
bstr1 = S.pack ("foo" :: String)

-- From overloaded string literal.
bstr2 :: S.ByteString
bstr2 = "bar"
```

See:

- Bytestring: Bits and Pieces
- ByteString

Printf

Haskell also has a variadic `printf` function in the style of C.

```
import Data.Text
import Text.Printf

a :: Int
a = 3

b :: Double
b = 3.14159
```

```
c :: String
c = "haskell"

example :: String
example = printf "(%i, %f, %s)" a b c
-- "(3, 3.14159, haskell)"
```

Overloaded Lists

It is ubiquitous for data structure libraries to expose `toList` and `fromList` functions to construct various structures out of lists. As of GHC 7.8 we now have the ability to overload the list syntax in the surface language with a typeclass `IsList`.

```
class IsList l where
  type Item l
  fromList :: [Item l] -> l
  toList   :: l -> [Item l]

instance IsList [a] where
  type Item [a] = a
  fromList = id
  toList   = id
```

```
λ: :type [1,2,3]
[1,2,3] :: (Num (Item l), IsList l) => l
```

```
{-# LANGUAGE OverloadedLists #-}
{-# LANGUAGE TypeFamilies #-}

import qualified Data.Map as Map
import GHC.Exts (IsList(..))

instance (Ord k) => IsList (Map.Map k v) where
  type Item (Map.Map k v) = (k,v)
  fromList = Map.fromList
  toList = Map.toList

example1 :: Map.Map String Int
example1 = [("a", 1), ("b", 2)]
```

Applicatives

Like monads Applicatives are an abstract structure for a wide class of computations that sit between functors and monads in terms of generality.

```haskell
pure  :: Applicative f => a -> f a
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<*>) :: f (a -> b) -> f a -> f b
```

As of GHC 7.6, Applicative is defined as:

```haskell
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

With the following laws:

```haskell
pure id <*> v = v
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
u <*> (v <*> w) = pure (.) <*> u <*> v <*> w
```

As an example, consider the instance for Maybe:

```haskell
instance Applicative Maybe where
  pure            = Just
  Nothing <*> _   = Nothing
  _ <*> Nothing   = Nothing
  Just f <*> Just x = Just (f x)
```

As a rule of thumb, whenever we would use `m >>= return . f` what we probably want is an applicative functor, and not a monad.

```haskell
import Network.HTTP
import Control.Applicative ((<$>),(<*>))

example1 :: Maybe Integer
example1 = (+) <$> m1 <*> m2
  where
    m1 = Just 3
    m2 = Nothing
-- Nothing

example2 :: [(Int, Int, Int)]
example2 = (,,) <$> m1 <*> m2 <*> m3
```

```haskell
  where
    m1 = [1,2]
    m2 = [10,20]
    m3 = [100,200]
 -- [(1,10,100),(1,10,200),(1,20,100),(1,20,200),(2,10,100),(2,10,200),(2,20,100),(2,20
```

```haskell
example3 :: IO String
example3 = (++) <$> fetch1 <*> fetch2
  where
    fetch1 = simpleHTTP (getRequest "http://www.fpcomplete.com/") >>= getResponseBody
    fetch2 = simpleHTTP (getRequest "http://www.haskell.org/") >>= getResponseBody
```

The pattern `f <$> a <*> b ...` shows up so frequently that there are a family of functions to lift applicatives of a fixed number arguments. This pattern also shows up frequently with monads (`liftM`, `liftM2`, `liftM3`).

```haskell
liftA :: Applicative f => (a -> b) -> f a -> f b
liftA f a = pure f <*> a

liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b

liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
liftA3 f a b c = f <$> a <*> b <*> c
```

Applicative also has functions `*>` and `<*` that sequence applicative actions while discarding the value of one of the arguments. The operator `*>` discard the left while `<*` discards the right. For example in a monadic parser combinator library the `*>` would parse with first parser argument but return the second.

The Applicative functions `<$>` and `<*>` are generalized by `liftM` and `ap` for monads.

```haskell
import Control.Monad
import Control.Applicative

data C a b = C a b

mnd :: Monad m => m a -> m b -> m (C a b)
mnd a b = C `liftM` a `ap` b

apl :: Applicative f => f a -> f b -> f (C a b)
apl a b = C <$> a <*> b
```

See: Applicative Programming with Effects

Typeclass Hierarchy

In principle every monad arises out of an applicative functor (and by corollary a functor) but due to historical reasons Applicative isn't a superclass of the Monad typeclass. A hypothetical fixed Prelude

might have:

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

class Applicative m => Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  ma >>= f = join (fmap f ma)

return :: Applicative m => a -> m a
return = pure


join :: Monad m => m (m a) -> m a
join x = x >>= id
```

See: <u>Functor-Applicative-Monad Proposal</u>

Alternative

Alternative is an extension of the Applicative class with a zero element and an associative binary operation respecting the zero.

```haskell
class Applicative f => Alternative f where
  -- | The identity of '<|>'
  empty :: f a
  -- | An associative binary operation
  (<|>) :: f a -> f a -> f a
  -- | One or more.
  some :: f a -> f [a]
  -- | Zero or more.
  many :: f a -> f [a]

optional :: Alternative f => f a -> f (Maybe a)
```

```haskell
instance Alternative Maybe where
    empty = Nothing
    Nothing <|> r = r
    l       <|> _ = l

instance Alternative [] where
    empty = []
    (<|>) = (++)
```

```
λ: foldl1 (<|>) [Nothing, Just 5, Just 3]
Just 5
```

These instances show up very frequently in parsers where the alternative operator can model alternative parse branches.

Polyvariadic Functions

One surprising application of typeclasses is the ability to construct functions which take an arbitrary number of arguments by defining instances over function types. The arguments may be of arbitrary type, but the resulting collected arguments must either converted into a single type or unpacked into a sum type.

```haskell
{-# LANGUAGE FlexibleInstances #-}

class Arg a where
  collect' :: [String] -> a

-- extract to IO
instance Arg (IO ()) where
  collect' acc = mapM_ putStrLn acc

-- extract to [String]
instance Arg [String] where
  collect' acc = acc

instance (Show a, Arg r) => Arg (a -> r) where
  collect' acc = \x -> collect' (acc ++ [show x])

collect :: Arg t => t
collect = collect' []

example1 :: [String]
example1 = collect 'a' 2 3.0

example2 :: IO ()
example2 = collect () "foo" [1,2,3]
```

See: Polyvariadic functions

Category

A category is an algebraic structure that includes a notion of an identity and a composition operation that is associative and preserves identities.

```haskell
class Category cat where
  id :: cat a a
  (.) :: cat b c -> cat a b -> cat a c
```

```haskell
instance Category (->) where
  id = Prelude.id
  (.) = (Prelude..)
```

```haskell
(<<<) :: Category cat => cat b c -> cat a b -> cat a c
(<<<) = (.)
```

```haskell
(>>>) :: Category cat => cat a b -> cat b c -> cat a c
f >>> g = g . f
```

Arrows

Arrows are an extension of categories with the notion of products.

```haskell
class Category a => Arrow a where
  arr :: (b -> c) -> a b c
  first :: a b c -> a (b,d) (c,d)
  second :: a b c -> a (d,b) (d,c)
  (***) :: a b c -> a b' c' -> a (b,b') (c,c')
  (&&&) :: a b c -> a b c' -> a b (c,c')
```

The canonical example is for functions.

```haskell
instance Arrow (->) where
  arr f = f
  first f = f *** id
  second f = id *** f
  (***) f g ~(x,y) = (f x, g y)
```

In this form functions of multiple arguments can be threaded around using the arrow combinators in a much more pointfree form. For instance a histogram function has a nice one-liner.

```haskell
histogram :: Ord a => [a] -> [(a, Int)]
histogram = map (head &&& length) . group . sort
```

```haskell
λ: histogram "Hello world"
[(' ',1),('H',1),('d',1),('e',1),('l',3),('o',2),('r',1),('w',1)]
```

Arrow notation

The following are equivalent:

```
{-# LANGUAGE Arrows #-}

addA :: Arrow a => a b Int -> a b Int -> a b Int
addA f g = proc x -> do
                y <- f -< x
                z <- g -< x
                returnA -< y + z
```

```
addA f g = arr (\ x -> (x, x)) >>>
           first f >>> arr (\ (y, x) -> (x, y)) >>>
           first g >>> arr (\ (z, y) -> y + z)
```

```
addA f g = f &&& g >>> arr (\ (y, z) -> y + z)
```

In practice this notation is not used often and in the future may become deprecated.

See: Arrow Notation

Bifunctors

Bifunctors are a generalization of functors to include types parameterized by two parameters and include two map functions for each parameter.

```
class Bifunctor p where
  bimap :: (a -> b) -> (c -> d) -> p a c -> p b d
  first :: (a -> b) -> p a c -> p b c
  second :: (b -> c) -> p a b -> p a c
```

The bifunctor laws are a natural generalization of the usual functor. Namely they respect identities and composition in the usual way:

```
bimap id id ≡ id
first id ≡ id
second id ≡ id
```

```
bimap f g ≡ first f . second g
```

The canonical example is for 2-tuples.

```
λ: first (+1) (1,2)
(2,2)
```

```
λ: second (+1) (1,2)
(1,3)
λ: bimap (+1) (+1) (1,2)
(2,3)

λ: first (+1) (Left 3)
Left 4
λ: second (+1) (Left 3)
Left 3
λ: second (+1) (Right 3)
Right 4
```

Error Handling

Control.Exception

The low-level (and most dangerous) way to handle errors is to use the `throw` and `catch` functions which allow us to throw extensible exceptions in pure code but catch the resulting exception within IO. Of specific note is that return value of the `throw` inhabits all types. There's no reason to use this for custom code that doesn't use low-level system operations.

```
throw :: Exception e => e -> a
catch :: Exception e => IO a -> (e -> IO a) -> IO a
try :: Exception e => IO a -> IO (Either e a)
evaluate :: a -> IO a
```

```
{-# LANGUAGE DeriveDataTypeable #-}

import Data.Typeable
import Control.Exception

data MyException = MyException
    deriving (Show, Typeable)

instance Exception MyException

evil :: [Int]
evil = [throw MyException]

example1 :: Int
example1 = head evil

example2 :: Int
example2 = length evil

main :: IO ()
main = do
  a <- try (evaluate example1) :: IO (Either MyException Int)
  print a
```

```
  b <- try (return example2) :: IO (Either MyException Int)
  print b
```

Because a value will not be evaluated unless needed, if one desires to know for sure that an exception is either caught or not it can be deeply forced into head normal form before invoking catch. The `strictCatch` is not provided by standard library but has a simple implementation in terms of `deepseq` .

```
strictCatch :: (NFData a, Exception e) => IO a -> (e -> IO a) -> IO a
strictCatch = catch . (toNF =<<)
```

Exceptions

The problem with the previous approach is having to rely on GHC's asynchronous exception handling inside of IO to handle basic operations. The `exceptions` provides the same API as `Control.Exception` but loosens the dependency on IO.

```
{-# LANGUAGE DeriveDataTypeable #-}

import Data.Typeable
import Control.Monad.Catch
import Control.Monad.Identity

data MyException = MyException
    deriving (Show, Typeable)

instance Exception MyException

example :: MonadCatch m => Int -> Int -> m Int
example x y | y == 0 = throwM MyException
            | otherwise = return $ x `div` y

pure :: MonadCatch m => m (Either MyException Int)
pure = do
  a <- try (example 1 2)
  b <- try (example 1 0)
  return (a >> b)
```

See: exceptions

Either

The instance of the Either monad is simple, note the bias toward Left when binding.

```
instance Monad (Either e) where
  return x = Right x

  (Left x)  >>= f = Left x
```

```
(Right x) >>= f = f x
```

The silly example one always sees is writing safe division function that fails out with a Left value when a division by zero happens and holds the resulting value in Right otherwise.

```haskell
sdiv :: Double -> Double -> Either String Double
sdiv _ 0 = throwError "divide by zero"
sdiv i j = return $ i / j

example :: Double -> Double -> Either String Double
example n m = do
  a <- sdiv n m
  b <- sdiv 2 a
  c <- sdiv 2 b
  return c

throwError :: String -> Either String b
throwError a = Left a

main :: IO ()
main = do
  print $ example 1 5
  print $ example 1 0
```

This is admittedly pretty stupid but captures the essence of why Either/EitherT is a suitable monad for exception handling.

ErrorT

In the monad transformer style, we can use the `ErrorT` transformer composed with an Identity monad and unrolling into an `Either Exception a`. This method is simple but requires manual instantiation of an Exception (or Typeable) typeclass if a custom Exception type is desired.

```haskell
import Control.Monad.Error
import Control.Monad.Identity

data Exception
  = Failure String
  | GenericFailure
  deriving Show

instance Error Exception where
  noMsg = GenericFailure

type ErrMonad a = ErrorT Exception Identity a

example :: Int -> Int -> ErrMonad Int
example x y = do
  case y of
    0 -> throwError $ Failure "division by zero"
```

```
            x -> return $ x `div` y

runFail :: ErrMonad a -> Either Exception a
runFail = runIdentity . runErrorT

example1 :: Either Exception Int
example1 = runFail $ example 2 3

example2 :: Either Exception Int
example2 = runFail $ example 2 0
```

ExceptT

As of mtl 2.2 or higher, the `ErrorT` class has been replaced by the `ExceptT` which fixes many of the problems with the old class.

At transformers level.

```
newtype ExceptT e m a = ExceptT (m (Either e a))

runExceptT :: ExceptT e m a -> m (Either e a)
runExceptT (ExceptT m) = m

instance (Monad m) => Monad (ExceptT e m) where
    return a = ExceptT $ return (Right a)
    m >>= k = ExceptT $ do
        a <- runExceptT m
        case a of
            Left e -> return (Left e)
            Right x -> runExceptT (k x)
    fail = ExceptT . fail

throwE :: (Monad m) => e -> ExceptT e m a
throwE = ExceptT . return . Left

catchE :: (Monad m) =>
    ExceptT e m a                -- ^ the inner computation
    -> (e -> ExceptT e' m a)     -- ^ a handler for exceptions in the inner
                                 -- computation
    -> ExceptT e' m a
m `catchE` h = ExceptT $ do
    a <- runExceptT m
    case a of
        Left  l -> runExceptT (h l)
        Right r -> return (Right r)
```

At MTL level.

```
instance MonadTrans (ExceptT e) where
    lift = ExceptT . liftM Right
```

```
class (Monad m) => MonadError e m | m -> e where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a

instance MonadError IOException IO where
    throwError = ioError
    catchError = catch

instance MonadError e (Either e) where
    throwError              = Left
    Left  l `catchError` h = h l
    Right r `catchError` _ = Right r
```

See:

- Control.Monad.Except

EitherT

```
newtype EitherT e m a = EitherT {runEitherT :: m (Either e a)}
        -- Defined in `Control.Monad.Trans.Either'
```

```
runEitherT :: EitherT e m a -> m (Either e a)
tryIO :: MonadIO m => IO a -> EitherT IOException m a

throwT  :: Monad m => e -> EitherT e m r
catchT  :: Monad m => EitherT a m r -> (a -> EitherT b m r) -> EitherT b m r
handleT :: Monad m => (a -> EitherT b m r) -> EitherT a m r -> EitherT b m
```

The ideal monad to use is simply the `EitherT` monad which we'd like to be able to use with an API similar to `ErrorT`. For example suppose we wanted to use `read` to attempt to read a positive integer from stdin. There are two failure modes and two failure cases here, one for a parse error which fails with an error from `Prelude.readIO` and one for a non-positive integer which fails with a custom exception after a check. We'd like to unify both cases in the same transformer.

Combined, the `safe` and `errors` make life with `EitherT` more pleasant. The safe library provides a variety of safer variants of the standard prelude functions that handle failures as Maybe values, explicitly passed default values, or more informative exception "notes". While the errors library reexports the safe Maybe functions and hoists them up into the `EitherT` monad providing a family of `try` prefixed functions that perform actions and can fail with an exception.

```
-- Exception handling equivalent of `read`
tryRead :: (Monad m, Read a) => e -> String -> EitherT e m a

-- Exception handling equivelent of `head`
tryHead :: Monad m => e -> [a] -> EitherT e m a

-- Exception handling equivelent of `(!!)`
```

```
tryAt :: Monad m => e -> [a] -> Int -> EitherT e m a
```

```
import Control.Error
import Control.Monad.Trans

data Failure
  = NonPositive Int
  | ReadError String
  deriving Show

main :: IO ()
main = do
  putStrLn "Enter a positive number."
  s <- getLine

  e <- runEitherT $ do
      n <- tryRead (ReadError s) s
      if n > 0
        then return $ n + 1
        else throwT $ NonPositive n

  case e of
      Left  n -> putStrLn $ "Failed with: " ++ show n
      Right s -> putStrLn $ "Succeeded with: " ++ show s
```

See:

- [Error Handling Simplified](#)
- [Safe](#)

Advanced Monads

Function Monad

If one writes Haskell long enough one might eventually encounter the curious beast that is the `((->) r)` monad instance. It generally tends to be non-intuitive to work with, but is quite simple when one considers it as an unwrapped Reader monad.

```
instance Functor ((->) r) where
  fmap = (.)

instance Monad ((->) r) where
  return = const
  f >>= k = \r -> k (f r) r
```

This just uses a prefix form of the arrow type operator.

```
import Control.Monad
```

```
id' :: (->) a a
id' = id

const' :: (->) a ((->) b a)
const' = const

-- Monad m => a -> m a
fret :: a -> b -> a
fret = return

-- Monad m => m a -> (a -> m b) -> m b
fbind :: (r -> a) -> (a -> (r -> b)) -> (r -> b)
fbind f k = f >>= k

-- Monad m => m (m a) -> m a
fjoin :: (r -> (r -> a)) -> (r -> a)
fjoin = join

fid :: a -> a
fid = const >>= id

-- Functor f => (a -> b) -> f a -> f b
fcompose :: (a -> b) -> (r -> a) -> (r -> b)
fcompose = (.)
```

```
type Reader r = (->) r -- pseudocode

instance Monad (Reader r) where
  return a = \_ -> a
  f >>= k = \ r -> k (f r) r

ask' :: r -> r
ask' = id

asks' :: (r -> a) -> (r -> a)
asks' f = id . f

runReader' :: (r -> a) -> r -> a
runReader' = id
```

RWS Monad

The RWS monad combines the functionality of the three monads discussed above, the **R**eader, **W**riter, and **S**tate. There is also a RWST transformer.

```
runReader :: Reader r a -> r -> a
runWriter :: Writer w a -> (a, w)
runState  :: State s a -> s -> (a, s)
```

These three eval functions are now combined into the following functions:

```
runRWS  :: RWS r w s a -> r -> s -> (a, s, w)
execRWS :: RWS r w s a -> r -> s -> (s, w)
evalRWS :: RWS r w s a -> r -> s -> (a, w)
```

```
import Control.Monad.RWS

type R = Int
type W = [Int]
type S = Int

computation :: RWS R W S ()
computation = do
  e <- ask
  a <- get
  let b = a + e
  put b
  tell [b]

example = runRWS computation 2 3
```

The usual caveat about Writer laziness also applies to RWS.

Cont

```
runCont :: Cont r a -> (a -> r) -> r
callCC :: MonadCont m => ((a -> m b) -> m a) -> m a
cont :: ((a -> r) -> r) -> Cont r a
```

In continuation passing style, composite computations are built up from sequences of nested computations which are terminated by a final continuation which yields the result of the full computation by passing a function into the continuation chain.

```
add :: Int -> Int -> Int
add x y = x + y
```

```
add :: Int -> Int -> (Int -> r) -> r
add x y k = k (x + y)
```

```
import Control.Monad
import Control.Monad.Cont
```

```haskell
add :: Int -> Int -> Cont k Int
add x y = return $ x + y

mult :: Int -> Int -> Cont k Int
mult x y = return $ x * y

contt :: ContT () IO ()
contt = do
    k <- do
       callCC $ \exit -> do
          lift $ putStrLn "Entry"
          exit $ \_ -> do
             putStrLn "Exit"
       lift $ putStrLn "Inside"
       lift $ k ()

callcc :: Cont String Integer
callcc = do
  a <- return 1
  b <- callCC (\k -> k 2)
  return $ a+b

ex1 :: IO ()
ex1 = print $ runCont (f >>= g) id
  where
     f = add 1 2
     g = mult 3
-- 9

ex2 :: IO ()
ex2 = print $ runCont callcc show
-- "3"

ex3 :: IO ()
ex3 = runContT contt print
-- Entry
-- Inside
-- Exit

main :: IO ()
main = do
  ex1
  ex2
  ex3
```

```haskell
newtype Cont r a = Cont { runCont :: ((a -> r) -> r) }

instance Monad (Cont r) where
  return a       = Cont $ \k -> k a
  (Cont c) >>= f = Cont $ \k -> c (\a -> runCont (f a) k)

class (Monad m) => MonadCont m where
```

```
  callCC :: ((a -> m b) -> m a) -> m a

instance MonadCont (Cont r) where
  callCC f = Cont $ \k -> runCont (f (\a -> Cont $ \_ -> k a)) k
```

- [Wikibooks: Continuation Passing Style](#)
- [MonadCont Under the Hood](#)

MonadPlus

Choice and failure.

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a

instance MonadPlus [] where
    mzero = []
    mplus = (++)

instance MonadPlus Maybe where
    mzero = Nothing

    Nothing `mplus` ys  = ys
    xs      `mplus` _ys = xs
```

MonadPlus forms a monoid with

```
mzero `mplus` a = a
a `mplus` mzero = a
(a `mplus` b) `mplus` c = a `mplus` (b `mplus` c)
```


```
when :: (Monad m) => Bool -> m () -> m ()
when p s =  if p then s else return ()

guard :: MonadPlus m => Bool -> m ()
guard True  = return ()
guard False = mzero

msum :: MonadPlus m => [m a] -> m a
msum =  foldr mplus mzero
```


```
import Safe
import Control.Monad

list1 :: [(Int,Int)]
```

```
list1 = [(a,b) | a <- [1..25], b <- [1..25], a < b]

list2 :: [(Int,Int)]
list2 = do
  a <- [1..25]
  b <- [1..25]
  guard (a < b)
  return $ (a,b)

maybe1 :: String -> String -> Maybe Double
maybe1 a b = do
  a' <- readMay a
  b' <- readMay b
  guard (b' /= 0.0)
  return $ a'/b'

maybe2 :: Maybe Int
maybe2 = msum [Nothing, Nothing, Just 3, Just 4]
```

```
import Control.Monad

range :: MonadPlus m => [a] -> m a
range [] = mzero
range (x:xs) = range xs `mplus` return x

pyth :: Integer -> [(Integer,Integer,Integer)]
pyth n = do
  x <- range [1..n]
  y <- range [1..n]
  z <- range [1..n]
  if x*x + y*y == z*z then return (x,y,z) else mzero

main :: IO ()
main = print $ pyth 15
{-
[ ( 12 , 9 , 15 )
, ( 12 , 5 , 13 )
, ( 9 , 12 , 15 )
, ( 8 , 6 , 10 )
, ( 6 , 8 , 10 )
, ( 5 , 12 , 13 )
, ( 4 , 3 , 5 )
, ( 3 , 4 , 5 )
]
-}
```

MonadFix

The fixed point of a monadic computation. `mfix f` executes the action `f` only once, with the eventual output fed back as the input.

```
fix :: (a -> a) -> a
fix f = let x = f x in x

mfix :: (a -> m a) -> m a
```

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

instance MonadFix Maybe where
    mfix f = let a = f (unJust a) in a
             where unJust (Just x) = x
                   unJust Nothing  = error "mfix Maybe: Nothing"
```

The regular do-notation can also be extended with `-XRecursiveDo` to accommodate recursive monadic bindings.

```
{-# LANGUAGE RecursiveDo #-}

import Control.Applicative
import Control.Monad.Fix

stream1 :: Maybe [Int]
stream1 = do
  rec xs <- Just (1:xs)
  return (map negate xs)

stream2 :: Maybe [Int]
stream2 = mfix $ \xs -> do
  xs' <- Just (1:xs)
  return (map negate xs')
```

ST Monad

The ST monad models "threads" of stateful computations which can manipulate mutable references but are restricted to only return pure values when evaluated and are statically confined to the ST monad of a `s` thread.

```
runST :: (forall s. ST s a) -> a
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

```
import Data.STRef
import Control.Monad
import Control.Monad.ST
```

```
import Control.Monad.State.Strict

example1 :: Int
example1 = runST $ do
  x <- newSTRef 0

  forM_ [1..1000] $ \j -> do
    writeSTRef x j

  readSTRef x

example2 :: Int
example2 = runST $ do
  count <- newSTRef 0
  replicateM_ (10^6) $ modifySTRef' count (+1)
  readSTRef count

example3 :: Int
example3 = flip evalState 0 $ do
  replicateM_ (10^6) $ modify' (+1)
  get

modify' :: MonadState a m => (a -> a) -> m ()
modify' f = get >>= (\x -> put $! f x)
```

Using the ST monad we can create a class of efficient purely functional data structures that use mutable references in a referentially transparent way.

Free Monads

```
Pure :: a -> Free f a
Free :: f (Free f a) -> Free f a

liftF :: (Functor f, MonadFree f m) => f a -> m a
retract :: Monad f => Free f a -> f a
```

Free monads are monads which instead of having a (join) operation that combines computations, instead forms composite computations from application of a functor.

```
join :: Monad m => m (m a) -> m a
wrap :: MonadFree f m => f (m a) -> m a
```

One of the best examples is the Partiality monad which models computations which can diverge. Haskell allows unbounded recursion, but for example we can create a free monad from the (Maybe) functor which can be used to fix the call-depth of, for example the Ackermann function.

```
import Control.Monad.Fix
```

```haskell
import Control.Monad.Free

type Partiality a = Free Maybe a

-- Non-termination.
never :: Partiality a
never = fix (Free . Just)

fromMaybe :: Maybe a -> Partiality a
fromMaybe (Just x) = Pure x
fromMaybe Nothing = Free Nothing

runPartiality :: Int -> Partiality a -> Maybe a
runPartiality 0 _ = Nothing
runPartiality _ (Pure a) = Just a
runPartiality _ (Free Nothing) = Nothing
runPartiality n (Free (Just a)) = runPartiality (n-1) a

ack :: Int -> Int -> Partiality Int
ack 0 n = Pure $ n + 1
ack m 0 = Free $ Just $ ack (m-1) 1
ack m n = Free $ Just $ ack m (n-1) >>= ack (m-1)

main :: IO ()
main = do
  let diverge = never :: Partiality ()
  print $ runPartiality 1000 diverge
  print $ runPartiality 1000 (ack 3 4)
  print $ runPartiality 5500 (ack 3 4)
```

The other common use for free monads is to build embedded domain-specific languages to describe computations. We can model a subset of the IO monad by building up a pure description of the computation inside of the IOFree monad and then using the free monad to encode the translation to an effectful IO computation.

```haskell
{-# LANGUAGE DeriveFunctor #-}

import System.Exit
import Control.Monad.Free

data Interaction x
  = Puts String x
  | Gets (Char -> x)
  | Exit
  deriving Functor

type IOFree a = Free Interaction a

puts :: String -> IOFree ()
puts s = liftF $ Puts s ()

get :: IOFree Char
```

```haskell
get = liftF $ Gets id

exit :: IOFree r
exit = liftF Exit

gets :: IOFree String
gets = do
  c <- get
  if c == '\n'
    then return ""
    else gets >>= \line -> return (c : line)

-- Collapse our IOFree DSL into IO monad actions.
interp :: IOFree a -> IO a
interp (Pure r) = return r
interp (Free x) = case x of
  Puts s t -> putStrLn s >> interp t
  Gets f   -> getChar >>= interp . f
  Exit     -> exitSuccess

echo :: IOFree ()
echo = do
  puts "Enter your name:"
  str <- gets
  puts str
  if length str > 10
    then puts "You have a long name."
    else puts "You have a short name."
  exit

main :: IO ()
main = interp echo
```

An implementation such as the one found in <u>free</u> might look like the following:

```haskell
{-# LANGUAGE MultiParamTypeClasses #-}

import Control.Applicative

data Free f a
  = Pure a
  | Free (f (Free f a))

instance Functor f => Monad (Free f) where
  return a      = Pure a
  Pure a >>= f = f a
  Free f >>= g = Free (fmap (>>= g) f)

class Monad m => MonadFree f m   where
  wrap :: f (m a) -> m a

liftF :: (Functor f, MonadFree f m) => f a -> m a
```

```
liftF = wrap . fmap return

iter :: Functor f => (f a -> a) -> Free f a -> a
iter _ (Pure a) = a
iter phi (Free m) = phi (iter phi <$> m)

retract :: Monad f => Free f a -> f a
retract (Pure a) = return a
retract (Free as) = as >>= retract
```

See:

- [Monads for Free!](#)
- [I/O is not a Monad](#)

Indexed Monads

Indexed monads are a generalisation of monads that adds an additional type parameter to the class that carries information about the computation or structure of the monadic implementation.

```
class IxMonad md where
  return :: a -> md i i a
  (>>=) :: md i m a -> (a -> md m o b) -> md i o b
```

The canonical use-case is a variant of the vanilla State which allows type-changing on the state for intermediate steps inside of the monad. This indeed turns out to be very useful for handling a class of problems involving resource management since the extra index parameter gives us space to statically enforce the sequence of monadic actions by allowing and restricting certain state transitions on the index parameter at compile-time.

To make this more usable we'll use the somewhat esoteric `-XRebindableSyntax` allowing us to overload the do-notation and if-then-else syntax by providing alternative definitions local to the module.

```
{-# LANGUAGE RebindableSyntax #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE NoMonomorphismRestriction #-}

import Data.IORef
import Data.Char
import Prelude hiding (fmap, (>>=), (>>), return)
import Control.Applicative

newtype IState i o a = IState { runIState :: i -> (a, o) }

evalIState :: IState i o a -> i -> a
evalIState st i = fst $ runIState st i

execIState :: IState i o a -> i -> o
execIState st i = snd $ runIState st i

ifThenElse :: Bool -> a -> a -> a
```

```haskell
ifThenElse b i j = case b of
  True -> i
  False -> j

return :: a -> IState s s a
return a = IState $ \s -> (a, s)

fmap :: (a -> b) -> IState i o a -> IState i o b
fmap f v = IState $ \i -> let (a, o) = runIState v i
                          in (f a, o)

join :: IState i m (IState m o a) -> IState i o a
join v = IState $ \i -> let (w, m) = runIState v i
                        in runIState w m

(>>=) :: IState i m a -> (a -> IState m o b) -> IState i o b
v >>= f = IState $ \i -> let (a, m) = runIState v i
                         in runIState (f a) m

(>>) :: IState i m a -> IState m o b -> IState i o b
v >> w = v >>= \_ -> w

get :: IState s s s
get = IState $ \s -> (s, s)

gets :: (a -> o) -> IState a o a
gets f = IState $ \s -> (s, f s)

put :: o -> IState i o ()
put o = IState $ \_ -> ((), o)

modify :: (i -> o) -> IState i o ()
modify f = IState $ \i -> ((), f i)



data Locked = Locked
data Unlocked = Unlocked

type Stateful a = IState a Unlocked a

acquire :: IState i Locked ()
acquire = put Locked

-- Can only release the lock if it's held, try release the lock
-- that's not held is a now a type error.
release :: IState Locked Unlocked ()
release = put Unlocked

-- Statically forbids improper handling of resources.
lockExample :: Stateful a
lockExample = do ptr <- get   :: IState a a a
                 acquire      :: IState a Locked ()
                     -- ...
```

```
                release   :: IState Locked Unlocked ()
                return ptr

-- Couldn't match type 'Locked' with 'Unlocked'
-- In a stmt of a 'do' block: return ptr
failure1 :: Stateful a
failure1 = do ptr <- get
              acquire
              return ptr -- didn't release


-- Couldn't match type 'a' with 'Locked'
-- In a stmt of a 'do' block: release
failure2 :: Stateful a
failure2 = do ptr <- get
              release -- didn't acquire
              return ptr


-- Evaluate the resulting state, statically ensuring that the
-- lock is released when finished.
evalReleased :: IState i Unlocked a -> i -> a
evalReleased f st = evalIState f st


example :: IO (IORef Integer)
example = evalReleased <$> pure lockExample <*> newIORef 0
```

See: Fun with Indexed monads

Quantification

Universal Quantification

Universal quantification the primary mechanism of encoding polymorphism in Haskell. The essence of universal quantification is that we can express functions which operate the same way for a set of types and whose function behavior is entirely determined *only* by the behavior of all types in this span.

```
{-# LANGUAGE ExplicitForAll #-}

-- ∀a. [a]
example1 :: forall a. [a]
example1 = []


-- ∀a. [a]
example2 :: forall a. [a]
example2 = [undefined]


-- ∀a. ∀b. (a → b) → [a] → [b]
map' :: forall a. forall b. (a -> b) -> [a] -> [b]
map' f = foldr ((:) . f) []


-- ∀a. [a] → [a]
reverse' :: forall a. [a] -> [a]
reverse' = foldl (flip (:)) []
```

Normally quantifiers are omitted in type signatures since in Haskell's vanilla surface language it is unambiguous to assume to that free type variables are universally quantified.

Free theorems

A universally quantified type-variable actually implies quite a few rather deep properties about the implementation of a function that can be deduced from its type signature. For instance the identity function in Haskell is guaranteed to only have one implementation since the only information that the information that can present in the body

```
id :: forall a. a -> a
id x = x
```

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

The free theorem of fmap:

```
forall f g. fmap f . fmap g = fmap (f . g)
```

See: Theorems for Free

Type Systems

Hindley Milner Typesystem

The Hindley-Milner typesystem is historically important as one of the first typed lambda calculi that admitted both polymorphism and a variety of inference techniques that could always decide principle types.

```
e : x
  | λx:t.e           -- value abstraction
  | e1 e2            -- application
  | let x = e1 in e2  -- let

t : t -> t     -- function types
  | a          -- type variables

σ : ∀ a . t    -- type scheme
```

In an implementation, the function `generalize` converts all type variables within the type into polymorphic type variables yielding a type scheme. The function `instantiate` maps a scheme to a type, but with any polymorphic variables converted into unbound type variables.

Rank-N Types

System-F is the type system that underlies Haskell. System-F subsumes the HM type system in the

sense that every type expressible in HM can be expressed within System-F. System-F is sometimes referred to in texts as the *Girald-Reynolds polymorphic lambda calculus* or *second-order lambda calculus*.

```
t : t -> t        -- function types
   | a            -- type variables
   | ∀ a . t      -- forall

e : x             -- variables
   | λ(x:t).e     -- value abstraction
   | e1 e2        -- value application
   | Λa.e         -- type abstraction
   | e_t          -- type application
```

An example with equivalents of GHC Core in comments:

```
id : ∀ t. t -> t
id = Λt. λx:t. x
-- id :: forall t. t -> t
-- id = \ (@ t) (x :: t) -> x

tr : ∀ a. ∀ b. a -> b -> a
tr = Λa. Λb. λx:a. λy:b. x
-- tr :: forall a b. a -> b -> a
-- tr = \ (@ a) (@ b) (x :: a) (y :: b) -> x

fl : ∀ a. ∀ b. a -> b -> b
fl = Λa. Λb. λx:a. λy:b. y
-- fl :: forall a b. a -> b -> b
-- fl = \ (@ a) (@ b) (x :: a) (y :: b) -> y

nil : ∀ a. [a]
nil = Λa. Λb. λz:b. λf:(a -> b -> b). z
-- nil :: forall a. [a]
-- nil = \ (@ a) (@ b) (z :: b) (f :: a -> b -> b) -> z

cons : ∀ a. a -> [a] -> [a]
cons = Λa. λx:a. λxs:(∀ b. b -> (a -> b -> b) -> b).
       Λb. λz:b. λf : (a -> b -> b). f x (xs_b z f)
-- cons :: forall a. a
--         -> (forall b. (a -> b -> b) -> b) -> (forall b. (a -> b -> b) -> b)
-- cons = \ (@ a) (x :: a) (xs :: forall b. (a -> b -> b) -> b)
--     (@ b) (z :: b) (f :: a -> b -> b) -> f x (xs @ b z f)
```

Normally when Haskell's typechecker infers a type signature it places all quantifiers of type variables at the outermost position such that no quantifiers appear within the body of the type expression, called the prenex restriction. This restricts an entire class of type signatures that would otherwise be expressible within System-F, but has the benefit of making inference much easier.

-XRankNTypes loosens the prenex restriction such that we may explicitly place quantifiers within the body of the type. The bad news is that the general problem of inference in this relaxed system is

undecidable in general, so we're required to explicitly annotate functions which use RankNTypes or they are otherwise inferred as rank 1 and may not typecheck at all.

```
{-# LANGUAGE RankNTypes #-}

-- Can't unify ( Bool ~ Char )
rank1 :: forall a. (a -> a) -> (Bool, Char)
rank1 f = (f True, f 'a')

rank2 :: (forall a. a -> a) -> (Bool, Char)
rank2 f = (f True, f 'a')

auto :: (forall a. a -> a) -> (forall b. b -> b)
auto x = x

xauto :: forall a. (forall b. b -> b) -> a -> a
xauto f = f
```

```
Monomorphic Rank 0: t
Polymorphic Rank 1: forall a. a -> t
Polymorphic Rank 2: (forall a. a -> t) -> t
Polymorphic Rank 3: ((forall a. a -> t) -> t) -> t
```

Of important note is that the type variables bound by an explicit quantifier in a higher ranked type may not escape their enclosing scope, the typechecker will explicitly enforce this with by enforcing that variables bound inside of rank-n types (called skolem constants) will not unify with free meta type variables inferred by the inference engine.

```
{-# LANGUAGE RankNTypes #-}

escape :: (forall a. a -> a) -> Int
escape f = f 0

g x = escape (\a -> x)
```

In this example in order for the expression to be well typed, ⟨f⟩ would necessarily have (⟨Int -> Int⟩) which implies that ⟨a ˜ Int⟩ over the whole type, but since ⟨a⟩ is bound under the quantifier it must not be unified with ⟨Int⟩ and so the typechecker must fail with a skolem capture error.

```
Couldn't match expected type 'a' with actual type 't'
'a' is a rigid type variable bound by a type expected by the context: a -> a
't' is a rigid type variable bound by the inferred type of g :: t -> Int
In the expression: x In the first argument of 'escape', namely '(\ a -> x)'
In the expression: escape (\ a -> x)
```

This can actually be used for our advantage to enforce several types of invariants about scope and use

of specific type variables. For example the ST monad uses a second rank type to prevent the capture of references between ST monads with separate state threads where the (s) type variable is bound within a rank-2 type and cannot escape, statically guaranteeing that the implementation details of the ST internals can't leak out and thus ensuring its referential transparency.

Existential Quantification

The essence of universal quantification is that we can express functions which operate the same way for *any* type, while for existential quantification we can express functions that operate over an *some* unknown type. Using an existential we can group heterogeneous values together with functions under the existential, that manipulate the data types but whose type signature hides this information.

```haskell
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE RankNTypes #-}

-- ∃ t. (t, t → t, t → String)
data Box = forall a. Box a (a -> a) (a -> String)

boxa :: Box
boxa = Box 1 negate show

boxb :: Box
boxb = Box "foo" reverse show

apply :: Box -> String
apply (Box x f p) = p (f x)

-- ∃ t. Show t => t
data SBox = forall a. Show a => SBox a

boxes :: [SBox]
boxes = [SBox (), SBox 2, SBox "foo"]

showBox :: SBox -> String
showBox (SBox a) = show a

main :: IO ()
main = mapM_ (putStrLn . showBox) boxes
-- ()
-- 2
-- "foo"
```

The existential over (SBox) gathers a collection of values defined purely in terms of their their Show interface, no other information is available about the values and they can't be accessed or unpacked in any other way.

```haskell
{-# LANGUAGE RankNTypes #-}

-- The functor is a fixed implementation of the library internals.
type Exists a b = forall f. Functor f => (b -> f b) -> (a -> f a)

type Get a b = a -> b
```

```
type Set a b = a -> b -> a

example :: Get a b -> Set a b -> Exists a b
example f g l a = fmap (g a) (l (f a))
```

Use of existentials can be used to recreate certain concepts from the so-called "Object Oriented Paradigm", a school of thought popularized in the late 80s that attempted to decompose programming logic into anthropomorphic entities and actions instead of the modern equational treatment. Recreating this model in Haskell is widely considered to be an antipattern.

See: Haskell Antipattern: Existential Typeclass

Impredicative Types

Although extremely brittle, GHC also has limited support impredicative polymorphism which allows instantiating type variable with a polymorphic type. Implied is that this loosens the restriction that quantifiers must precede arrow types and now they may be placed inside of type-constructors.

```
-- Can't unify ( Int ~ Char )

revUni :: forall a. Maybe ([a] -> [a]) -> Maybe ([Int], [Char])
revUni (Just g) = Just (g [3], g "hello")
revUni Nothing  = Nothing
```

```
{-# LANGUAGE ImpredicativeTypes #-}

-- Uses higher-ranked polymorphism.
f :: (forall a. [a] -> a) -> (Int, Char)
f get = (get [1,2], get ['a', 'b', 'c'])

-- Uses impredicative polymorphism.
g :: Maybe (forall a. [a] -> a) -> (Int, Char)
g Nothing = (0, '0')
g (Just get) = (get [1,2], get ['a','b','c'])
```

Use of this extension is very rare, and there is some consideration that `-XImpredicativeTypes` is fundamentally broken. Although GHC is very liberal about telling us to enable it when one accidentally makes a typo in a type signature!

Some notable trivia, the `($)` operator is wired into GHC in a very special way as to allow impredicative instantiation of `runST` to be applied via `($)` by special-casing the `($)` operator only when used for the ST monad. If this sounds like an ugly hack it's because it is, but a rather convenient hack.

For example if we define a function `apply` which should behave identically to `($)` we'll get an error about polymorphic instantiation even though they are defined identically!

```
{-# LANGUAGE RankNTypes #-}

import Control.Monad.ST
```

```
f `apply` x =  f x

foo :: (forall s. ST s a) -> a
foo st = runST $ st

bar :: (forall s. ST s a) -> a
bar st = runST `apply` st
```

```
    Couldn't match expected type `forall s. ST s a'
              with actual type `ST s0 a'
    In the second argument of `apply', namely `st'
    In the expression: runST `apply` st
    In an equation for `bar': bar st = runST `apply` st
```

See:

- SPJ Notes on $

Scoped Type Variables

Normally the type variables used within the toplevel signature for a function are only scoped to the type-signature and not the body of the function and its rigid signatures over terms and let/where clauses. Enabling `-XScopedTypeVariables` loosens this restriction allowing the type variables mentioned in the toplevel to be scoped within the value-level body of a function and all signatures contained therein.

```
{-# LANGUAGE ExplicitForAll #-}
{-# LANGUAGE ScopedTypeVariables #-}

poly :: forall a b c. a -> b -> c -> (a, a)
poly x y z = (f x y, f x z)
  where
    -- second argument is universally quantified from inference
    -- f :: forall t0 t1. t0 -> t1 -> t0
    f x' _ = x'

mono :: forall a b c. a -> b -> c -> (a, a)
mono x y z = (f x y, f x z)
  where
    -- b is not implictly universally quantified because it is in scope
    f :: a -> b -> a
    f x' _ = x'

example :: IO ()
example = do
  x :: [Int] <- readLn
  print x
```

GADTs

GADTs

Generalized Algebraic Data types (GADTs) are an extension to algebraic datatypes that allow us to qualify the constructors to datatypes with type equality constraints, allowing a class of types that are not expressible using vanilla ADTs.

`-XGADTs` implicitly enables an alternative syntax for datatype declarations (`-XGADTSyntax`) such that the following declarations are equivalent:

```haskell
-- Vanilla
data List a
  = Empty
  | Cons a (List a)

-- GADTSyntax
data List a where
  Empty :: List a
  Cons :: a -> List a -> List a
```

For an example use consider the data type `Term` , we have a term in which we `Succ` which takes a `Term` parameterized by `a` which span all types. Problems arise between the clash whether (`a ~ Bool`) or (`a ~ Int`) when trying to write the evaluator.

```haskell
data Term a
  = Lit a
  | Succ (Term a)
  | IsZero (Term a)

-- can't be well-typed :(
eval (Lit i)      = i
eval (Succ t)     = 1 + eval t
eval (IsZero i)   = eval i == 0
```

And we admit the construction of meaningless terms which forces more error handling cases.

```haskell
-- This is a valid type.
failure = Succ ( Lit True )
```

Using a GADT we can express the type invariants for our language (i.e. only type-safe expressions are representable). Pattern matching on this GADTs then carries type equality constraints without the need for explicit tags.

```haskell
{-# Language GADTs #-}

data Term a where
  Lit    :: a -> Term a
  Succ   :: Term Int -> Term Int
```

```
    IsZero :: Term Int -> Term Bool
    If     :: Term Bool -> Term a -> Term a -> Term a

eval :: Term a -> a
eval (Lit i)      = i                               -- Term a
eval (Succ t)     = 1 + eval t                      -- Term (a ~ Int)
eval (IsZero i)   = eval i == 0                     -- Term (a ~ Int)
eval (If b e1 e2) = if eval b then eval e1 else eval e2 -- Term (a ~ Bool)


example :: Int
example = eval (Succ (Succ (Lit 3)))
```

This time around:

```
-- This is rejected at compile-time.
failure = Succ ( Lit True )
```

Explicit equality constraints ($a \sim b$) can be added to a function's context. For example the following expand out to the same types.

```
f :: a -> a -> (a, a)
f :: (a ~ b) => a -> b -> (a,b)
```

```
(Int ~ Int)  => ...
(a ~ Int)    => ...
(Int ~ a)    => ...
(a ~ b)      => ...
(Int ~ Bool) => ...   -- Will not typecheck.
```

This is effectively the implementation detail of what GHC is doing behind the scenes to implement GADTs (implicitly passing and threading equality terms around). If we wanted we could do the same setup that GHC does just using equality constraints and existential quantification.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE ExistentialQuantification #-}

-- Using Constraints
data Exp a
  = (a ~ Int) => LitInt a
  | (a ~ Bool) => LitBool a
  | forall b. (b ~ Bool) => If (Exp b) (Exp a) (Exp a)

-- Using GADTs
-- data Exp a where
--   LitInt  :: Int  -> Exp Int
--   LitBool :: Bool -> Exp Bool
```

```
--    If        :: Exp Bool -> Exp a -> Exp a -> Exp a

eval :: Exp a -> a
eval e = case e of
  LitInt i   -> i
  LitBool b  -> b
  If b tr fl -> if eval b then eval tr else eval fl
```

In the presence of GADTs inference becomes intractable in many cases, often requiring an explicit annotation. For example `f` can either have `T a -> [a]` or `T a -> [Int]` and neither is principle.

```
data T :: * -> * where
  T1 :: Int -> T Int
  T2 :: T a

f (T1 n) = [n]
f T2     = []
```

Kind Signatures

Haskell's kind system (i.e. the "type of the types") is a system consisting the single kind `*` and an arrow kind `->`.

```
κ : *
  | κ -> κ
```

```
Int :: *
Maybe :: * -> *
Either :: * -> * -> *
```

There are in fact some extensions to this system that will be covered later (see: PolyKinds and Unboxed types in later sections) but most kinds in everyday code are simply either stars or arrows.

With the KindSignatures extension enabled we can now annotate top level type signatures with their explicit kinds, bypassing the normal kind inference procedures.

```
{-# LANGUAGE KindSignatures #-}

id :: forall (a :: *). a -> a
id x = x
```

On top of default GADT declaration we can also constrain the parameters of the GADT to specific kinds. For basic usage Haskell's kind inference can deduce this reasonably well, but combined with some other type system extensions that extend the kind system this becomes essential.

```haskell
{-# Language GADTs #-}
{-# LANGUAGE KindSignatures #-}

data Term a :: * where
  Lit    :: a -> Term a
  Succ   :: Term Int -> Term Int
  IsZero :: Term Int -> Term Bool
  If     :: Term Bool -> Term a -> Term a -> Term a

data Vec :: * -> * -> * where
  Nil :: Vec n a
  Cons :: a -> Vec n a -> Vec n a

data Fix :: (* -> *) -> * where
  In :: f (Fix f) -> Fix f
```

Void

The Void type is the type with no inhabitants. It unifies only with itself.

Using a newtype wrapper we can create a type where recursion makes it impossible to construct an inhabitant.

```haskell
-- Void :: Void -> Void
newtype Void = Void Void
```

Or using `-XEmptyDataDecls` we can also construct the uninhabited type equivalently as a data declaration with no constructors.

```haskell
data Void
```

The only inhabitant of both of these types is a diverging term like (`undefined`).

Phantom Types

Phantom types are parameters that appear on the left hand side of a type declaration but which are not constrained by the values of the types inhabitants. They are effectively slots for us to encode additional information at the type-level.

```haskell
import Data.Void

data Foo tag a = Foo a

combine :: Num a => Foo tag a -> Foo tag a -> Foo tag a
combine (Foo a) (Foo b) = Foo (a+b)

-- All identical at the value level, but differ at the type level.
```

```
a :: Foo () Int
a = Foo 1

b :: Foo t Int
b = Foo 1

c :: Foo Void Int
c = Foo 1

-- () ~ ()
example1 :: Foo () Int
example1 = combine a a

-- t ~ ()
example2 :: Foo () Int
example2 = combine a b

-- t0 ~ t1
example3 :: Foo t Int
example3 = combine b b

-- Couldn't match type 't' with 'Void'
example4 :: Foo t Int
example4 = combine b c
```

Notice the type variable (tag) does not appear in the right hand side of the declaration. Using this allows us to express invariants at the type-level that need not manifest at the value-level. We're effectively programming by adding extra information at the type-level.

See: Fun with Phantom Types

Type Equality

With a richer language for datatypes we can express terms that witness the relationship between terms in the constructors, for example we can now express a term which expresses propositional equality between two types.

The type (Eql a b) is a proof that types (a) and (b) are equal, by pattern matching on the single (Refl) constructor we introduce the equality constraint into the body of the pattern match.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE ExplicitForAll #-}

-- a ≡ b
data Eql a b where
  Refl :: Eql a a

-- Congruence
-- (f : A → B) {x y} → x ≡ y → f x ≡ f y
cong :: Eql a b -> Eql (f a) (f b)
cong Refl = Refl

-- Symmetry
```

```haskell
-- {a b : A} → a ≡ b → a ≡ b
sym :: Eql a b -> Eql b a
sym Refl = Refl


-- Transitivity
-- {a b c : A} → a ≡ b → b ≡ c → a ≡ c
trans :: Eql a b -> Eql b c -> Eql a c
trans Refl Refl = Refl


-- Coerce one type to another given a proof of their equality.
-- {a b : A} → a ≡ b → a → b
castWith :: Eql a b -> a -> b
castWith Refl = id


-- Trivial cases
a :: forall n. Eql n n
a = Refl


b :: forall. Eql () ()
b = Refl
```

As of GHC 7.8 these constructors and functions are included in the Prelude in the Data.Type.Equality module.

Interpreters

The lambda calculus forms the theoretical and practical foundation for many languages. At the heart of every calculus is three components:

- **Var** - A variable
- **Lam** - A lambda abstraction
- **App** - An application
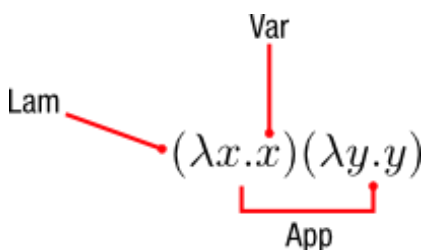
There are many different ways of modeling these constructions and data structure representations, but they all more or less contain these three elements. For example, a lambda calculus that uses String names on lambda binders and variables might be written like the following:

```haskell
type Name = String


data Exp
  = Var Name
  | Lam Name Exp
  | App Exp Exp
```

A lambda expression in which all variables that appear in the body of the expression are referenced in

an outer lambda binder is said to be *closed* while an expression with unbound free variables is *open*.

See: Mogensen–Scott encoding

HOAS

Higher Order Abstract Syntax (*HOAS*) is a technique for implementing the lambda calculus in a language where the binders of the lambda expression map directly onto lambda binders of the host language (i.e. Haskell) to give us substitution machinery in our custom language by exploiting Haskell's implementation.

```haskell
{-# LANGUAGE GADTs #-}

data Expr a where
  Con :: a -> Expr a
  Lam :: (Expr a -> Expr b) -> Expr (a -> b)
  App :: Expr (a -> b) -> Expr a -> Expr b

i :: Expr (a -> a)
i = Lam (\x -> x)

k :: Expr (a -> b -> a)
k = Lam (\x -> Lam (\y -> x))

s :: Expr ((a -> b -> c) -> (a -> b) -> (a -> c))
s = Lam (\x -> Lam (\y -> Lam (\z -> App (App x z) (App y z))))

eval :: Expr a -> a
eval (Con v) = v
eval (Lam f) = \x -> eval (f (Con x))
eval (App e1 e2) = (eval e1) (eval e2)


skk :: Expr (a -> a)
skk = App (App s k) k

example :: Integer
example = eval skk 1
-- 1
```

Pretty printing HOAS terms can also be quite complicated since the body of the function is under a Haskell lambda binder.

PHOAS

A slightly different form of HOAS called PHOAS uses lambda datatype parameterized over the binder type. In this form evaluation requires unpacking into a separate Value type to wrap the lambda expression.

```haskell
{-# LANGUAGE RankNTypes #-}

data ExprP a
```

```
    = VarP a
    | AppP (ExprP a) (ExprP a)
    | LamP (a -> ExprP a)
    | LitP Integer

data Value
    = VLit Integer
    | VFun (Value -> Value)

fromVFun :: Value -> (Value -> Value)
fromVFun val = case val of
   VFun f -> f
   _        -> error "not a function"

fromVLit :: Value -> Integer
fromVLit val = case val of
   VLit n -> n
   _        -> error "not a integer"

newtype Expr = Expr { unExpr :: forall a . ExprP a }

eval :: Expr -> Value
eval e = ev (unExpr e) where
   ev (LamP f)      = VFun(ev . f)
   ev (VarP v)      = v
   ev (AppP e1 e2)  = fromVFun (ev e1) (ev e2)
   ev (LitP n)      = VLit n

i :: ExprP a
i = LamP (\a -> VarP a)

k :: ExprP a
k = LamP (\x -> LamP (\y -> VarP x))

s :: ExprP a
s = LamP (\x -> LamP (\y -> LamP (\z -> AppP (AppP (VarP x) (VarP z)) (AppP (VarP y) (

skk :: ExprP a
skk = AppP (AppP s k) k

example :: Integer
example = fromVLit $ eval $ Expr (AppP skk (LitP 3))
```

See:

- PHOAS
- Encoding Higher-Order Abstract Syntax with Parametric Polymorphism

Final Interpreters

Using typeclasses we can implement a *final interpreter* which models a set of extensible terms using functions bound to typeclasses rather than data constructors. Instances of the typeclass form interpreters over these terms.

For example we can write a small language that includes basic arithmetic, and then retroactively extend our expression language with a multiplication operator without changing the base. At the same time our interpreter logic remains invariant under extension with new expressions.

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE NoMonomorphismRestriction #-}

class Expr repr where
  lit :: Int -> repr
  neg :: repr -> repr
  add :: repr -> repr -> repr
  mul :: repr -> repr -> repr

instance Expr Int where
  lit n = n
  neg a = -a
  add a b = a + b
  mul a b = a * b

instance Expr String where
  lit n = show n
  neg a = "(-" ++ a ++ ")"
  add a b = "(" ++ a ++ " + " ++ b ++ ")"
  mul a b = "(" ++ a ++ " * " ++ b ++ ")"

class BoolExpr repr where
  eq :: repr -> repr -> repr
  tr :: repr
  fl :: repr

instance BoolExpr Int where
  eq a b = if a == b then tr else fl
  tr = 1
  fl = 0

instance BoolExpr String where
  eq a b = "(" ++ a ++ " == " ++ b ++ ")"
  tr = "true"
  fl = "false"

eval :: Int -> Int
eval = id

render :: String -> String
render = id

expr :: (BoolExpr repr, Expr repr) => repr
expr = eq (add (lit 1) (lit 2)) (lit 3)

result :: Int
result = eval expr
```

```
-- 1

string :: String
string = render expr
-- "((1 + 2) == 3)"
```

Finally Tagless

Writing an evaluator for the lambda calculus can likewise also be modeled with a final interpreter and a Identity functor.

```haskell
import Prelude hiding (id)

class Expr rep where
  lam :: (rep a -> rep b) -> rep (a -> b)
  app :: rep (a -> b) -> (rep a -> rep b)
  lit :: a -> rep a

newtype Interpret a = R { reify :: a }

instance Expr Interpret where
  lam f   = R $ reify . f . R
  app f a = R $ reify f $ reify a
  lit     = R

eval :: Interpret a -> a
eval e = reify e

e1 :: Expr rep => rep Int
e1 = app (lam (\x -> x)) (lit 3)

e2 :: Expr rep => rep Int
e2 = app (lam (\x -> lit 4)) (lam $ \x -> lam $ \y -> y)

example1 :: Int
example1 = eval e1
-- 3

example2 :: Int
example2 = eval e2
-- 4
```

See: Typed Tagless Interpretations and Typed Compilation

Datatypes

The usual hand-wavy of describing algebraic datatypes is to indicate the how natural correspondence between sum types, product types, and polynomial expressions arises.

```
data Void                      -- 0
data Unit     = Unit           -- 1
data Sum a b  = Inl a | Inr b  -- a + b
data Prod a b = Prod a b       -- a * b
type (->) a b = a -> b         -- b ^ a
```

Intuitively it follows the notion that the cardinality of set of inhabitants of a type can always be given as a function of the number of its holes. A product type admits a number of inhabitants as a function of the product (i.e. cardinality of the Cartesian product), a sum type as the sum of its holes and a function type as the exponential of the span of the domain and codomain.

```
-- 1 + A
data Maybe a = Nothing | Just a
```

Recursive types are correspond to infinite series of these terms.

```
-- pseudocode

-- µX. 1 + X
data Nat a = Z | S Nat
Nat a = µ a. 1 + a
      = 1 + (1 + (1 + ...))

-- µX. 1 + A * X
data List a = Nil | Cons a (List a)
List a = µ a. 1 + a * (List a)
       = 1 + a + a^2 + a^3 + a^4 ...

-- µX. A + A*X*X
data Tree a f = Leaf a | Tree a f f
Tree a = µ a. 1 + a * (List a)
       = 1 + a^2 + a^4 + a^6 + a^8 ...
```

See: Species and Functors and Types, Oh My!

F-Algebras

The *initial algebra* approach differs from the final interpreter approach in that we now represent our terms as algebraic datatypes and the interpreter implements recursion and evaluation occurs through pattern matching.

```
type Algebra f a = f a -> a
type Coalgebra f a = a -> f a
newtype Fix f = Fix { unFix :: f (Fix f) }

cata :: Functor f => Algebra f a -> Fix f -> a
```

```
ana  :: Functor f => Coalgebra f a -> a -> Fix f
hylo :: Functor f => Algebra f b -> Coalgebra f a -> a -> b
```

In Haskell a F-algebra is a functor `f a` together with a function `f a -> a`. A coalgebra reverses the function. For a functor `f` we can form its recursive unrolling using the recursive `Fix` newtype wrapper.

```
newtype Fix f = Fix { unFix :: f (Fix f) }

Fix :: f (Fix f) -> Fix f
unFix :: Fix f -> f (Fix f)
```

```
Fix f = f (f (f (f (f (f ( ... ))))))

newtype T b a = T (a -> b)

Fix (T a)
Fix T -> a
(Fix T -> a) -> a
(Fix T -> a) -> a -> a
...
```

In this form we can write down a generalized fold/unfold function that are datatype generic and written purely in terms of the recursing under the functor.

```
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix

ana :: Functor f => Coalgebra f a -> a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg
```

We call these functions *catamorphisms* and *anamorphisms*. Notice especially that the types of these two functions simply reverse the direction of arrows. Interpreted in another way they transform an algebra/coalgebra which defines a flat structure-preserving mapping between `Fix f` `f` into a function which either rolls or unrolls the fixpoint. What is particularly nice about this approach is that the recursion is abstracted away inside the functor definition and we are free to just implement the flat transformation logic!

For example a construction of the natural numbers in this form:

```
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE UndecidableInstances #-}

type Algebra f a = f a -> a
```

```haskell
type Coalgebra f a = a -> f a

newtype Fix f = Fix { unFix :: f (Fix f) }

-- catamorphism
cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix

-- anamorphism
ana :: Functor f => Coalgebra f a -> a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg

-- hylomorphism
hylo :: Functor f => Algebra f b -> Coalgebra f a -> a -> b
hylo f g = cata f . ana g

type Nat = Fix NatF
data NatF a = S a | Z deriving (Eq, Show)

instance Functor NatF where
  fmap f Z     = Z
  fmap f (S x) = S (f x)

plus :: Nat -> Nat -> Nat
plus n = cata phi where
  phi Z     = n
  phi (S m) = s m

times :: Nat -> Nat -> Nat
times n = cata phi where
  phi Z     = z
  phi (S m) = plus n m

int :: Nat -> Int
int = cata phi where
  phi Z     = 0
  phi (S f) = 1 + f

nat :: Integer -> Nat
nat = ana (psi Z S) where
  psi f _ 0 = f
  psi _ f n = f (n-1)

z :: Nat
z = Fix Z

s :: Nat -> Nat
s = Fix . S


type Str = Fix StrF
data StrF x = Cons Char x | Nil

instance Functor StrF where
```

```haskell
  fmap f (Cons a as) = Cons a (f as)
  fmap f Nil = Nil

nil :: Str
nil = Fix Nil

cons :: Char -> Str -> Str
cons x xs = Fix (Cons x xs)

str :: Str -> String
str = cata phi where
  phi Nil        = []
  phi (Cons x xs) = x : xs

str' :: String -> Str
str' = ana (psi Nil Cons) where
  psi f _ []     = f
  psi _ f (a:as) = f a as

map' :: (Char -> Char) -> Str -> Str
map' f = hylo g unFix
  where
    g Nil        = Fix Nil
    g (Cons a x) = Fix $ Cons (f a) x


type Tree a = Fix (TreeF a)
data TreeF a f = Leaf a | Tree a f f deriving (Show)

instance Functor (TreeF a) where
  fmap f (Leaf a) = Leaf a
  fmap f (Tree a b c) = Tree a (f b) (f c)

depth :: Tree a -> Int
depth = cata phi where
  phi (Leaf _)     = 0
  phi (Tree _ l r) = 1 + max l r


example1 :: Int
example1 = int (plus (nat 125) (nat 25))
-- 150
```

Or for example an interpreter for a small expression language that depends on a scoping dictionary.

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE UndecidableInstances #-}

import Control.Applicative
```

```haskell
import qualified Data.Map as M

type Algebra f a = f a -> a
type Coalgebra f a = a -> f a

newtype Fix f = Fix { unFix :: f (Fix f) }

cata :: Functor f => Algebra f a -> Fix f -> a
cata alg = alg . fmap (cata alg) . unFix

ana :: Functor f => Coalgebra f a -> a -> Fix f
ana coalg = Fix . fmap (ana coalg) . coalg

hylo :: Functor f => Algebra f b -> Coalgebra f a -> a -> b
hylo f g = cata f . ana g

type Id = String
type Env = M.Map Id Int

type Expr = Fix ExprF
data ExprF a
  = Lit Int
  | Var Id
  | Add a a
  | Mul a a
  deriving (Show, Eq, Ord, Functor)

deriving instance Eq (f (Fix f)) => Eq (Fix f)
deriving instance Ord (f (Fix f)) => Ord (Fix f)
deriving instance Show (f (Fix f)) => Show (Fix f)

eval :: M.Map Id Int -> Fix ExprF -> Maybe Int
eval env = cata phi where
  phi ex = case ex of
    Lit c   -> pure c
    Var i   -> M.lookup i env
    Add x y -> liftA2 (+) x y
    Mul x y -> liftA2 (*) x y

expr :: Expr
expr = Fix (Mul n (Fix (Add x y)))
  where
    n = Fix (Lit 10)
    x = Fix (Var "x")
    y = Fix (Var "y")

env :: M.Map Id Int
env = M.fromList [("x", 1), ("y", 2)]

compose :: (f (Fix f) -> c) -> (a -> Fix f) -> a -> c
compose x y = x . unFix . y

example :: Maybe Int
example = eval env expr
```

```
-- Just 30
```

What's especially nice about this approach is how naturally catamorphisms compose into efficient composite transformations.

```
compose :: Functor f => (f (Fix f) -> c) -> (a -> Fix f) -> a -> c
compose f g = f . unFix . g
```

- [Understanding F-Algebras](#)

recursion-schemes

The code from the F-algebra examples above is implemented in an off-the-shelf library called `recursion-schemes`.

```
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE DeriveFunctor #-}

import Data.Functor.Foldable

type Var = String

data Exp
  = Var Var
  | App Exp Exp
  | Lam [Var] Exp
  deriving Show

data ExpF a
  = VarF Var
  | AppF a a
  | LamF [Var] a
  deriving Functor

type instance Base Exp = ExpF

instance Foldable Exp where
  project (Var a)    = VarF a
  project (App a b)  = AppF a b
  project (Lam a b)  = LamF a b

instance Unfoldable Exp where
  embed (VarF a)     = Var a
  embed (AppF a b)   = App a b
  embed (LamF a b)   = Lam a b

fvs :: Exp -> [Var]
fvs = cata phi
  where phi (VarF a)   = [a]
        phi (AppF a b) = a ++ b
```

```
        phi (LamF a b) = foldr (filter . (/=)) a b
```

An example of usage:

```haskell
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE TypeSynonymInstances #-}

import Data.Traversable
import Control.Monad hiding (forM_, mapM, sequence)
import Prelude hiding (mapM)
import qualified Data.Map as M

newtype Fix (f :: * -> *) = Fix { outF :: f (Fix f) }

-- Catamorphism
cata :: Functor f => (f a -> a) -> Fix f -> a
cata f = f . fmap (cata f) . outF

-- Monadic catamorphism
cataM :: (Traversable f, Monad m) => (f a -> m a) -> Fix f -> m a
cataM f = f <=< mapM (cataM f) . outF

data ExprF r
  = EVar String
  | EApp r r
  | ELam r r
  deriving (Show, Eq, Ord, Functor)

type Expr = Fix ExprF

instance Show (Fix ExprF) where
  show (Fix f) = show f

instance Eq (Fix ExprF) where
  Fix x == Fix y = x == y

instance Ord (Fix ExprF) where
  compare (Fix x) (Fix y) = compare x y


mkApp :: Fix ExprF -> Fix ExprF -> Fix ExprF
mkApp x y = Fix (EApp x y)

mkVar :: String -> Fix ExprF
mkVar x = Fix (EVar x)

mkLam :: Fix ExprF -> Fix ExprF -> Fix ExprF
mkLam x y = Fix (ELam x y)

i :: Fix ExprF
```

```
i = mkLam (mkVar "x") (mkVar "x")

k :: Fix ExprF
k = mkLam (mkVar "x") $ mkLam (mkVar "y") $ (mkVar "x")

subst :: M.Map String (ExprF Expr) -> Expr -> Expr
subst env = cata alg where
  alg (EVar x) | Just e <- M.lookup x env = Fix e
  alg e = Fix e
```

See:

- recursion-schemes

Hint and Mueval

GHC itself can actually interpret arbitrary Haskell source on the fly by hooking into the GHC's bytecode interpreter (the same used for GHCi). The hint package allows us to parse, typecheck, and evaluate arbitrary strings into arbitrary Haskell programs and evaluate them.

```
import Language.Haskell.Interpreter

foo :: Interpreter String
foo = eval "(\\x -> x) 1"

example :: IO (Either InterpreterError String)
example = runInterpreter foo
```

This is generally not a wise thing to build a library around, unless of course the purpose of the program is itself to evaluate arbitrary Haskell code (something like an online Haskell shell or the likes).

Both hint and mueval do effectively the same task, designed around slightly different internals of the GHC Api.

See:

- hint
- mueval

Testing

Contrary to a lot of misinformation, unit testing in Haskell is quite common and robust. Although generally speaking unit tests tend to be of less importance in Haskell since the type system makes an enormous amount of invalid programs completely inexpressible by construction. Unit tests tend to be written later in the development lifecycle and generally tend to be about the core logic of the program and not the intermediate plumbing.

A prominent school of thought on Haskell library design tends to favor constructing programs built around strong equation laws which guarantee strong invariants about program behavior under composition. Many of the testing tools are built around this style of design.

QuickCheck

Probably the most famous Haskell library, QuickCheck is a testing framework for generating large random tests for arbitrary functions automatically based on the types of their arguments.

```haskell
quickCheck :: Testable prop => prop -> IO ()
(==>) :: Testable prop => Bool -> prop -> Property
forAll :: (Show a, Testable prop) => Gen a -> (a -> prop) -> Property
choose :: Random a => (a, a) -> Gen a
```

```haskell
import Test.QuickCheck

qsort :: [Int] -> [Int]
qsort []     = []
qsort (x:xs) = qsort lhs ++ [x] ++ qsort rhs
    where lhs = filter  (< x) xs
          rhs = filter (>= x) xs

prop_maximum ::  [Int] -> Property
prop_maximum xs = not (null xs) ==>
                    last (qsort xs) == maximum xs

main :: IO ()
main = quickCheck prop_maximum
```

```
$ runhaskell qcheck.hs
*** Failed! Falsifiable (after 3 tests and 4 shrinks):
[0]
[1]

$ runhaskell qcheck.hs
+++ OK, passed 1000 tests.
```

The test data generator can be extended with custom types and refined with predicates that restrict the domain of cases to test.

```haskell
import Test.QuickCheck

data Color = Red | Green | Blue deriving Show

instance Arbitrary Color where
  arbitrary = do
    n <- choose (0,2) :: Gen Int
    return $ case n of
      0 -> Red
      1 -> Green
      2 -> Blue

example1 :: IO [Color]
example1 = sample' arbitrary
-- [Red,Green,Red,Blue,Red,Red,Red,Blue,Green,Red,Red]
```

See:

SmallCheck

Like QuickCheck, SmallCheck is a property testing system but instead of producing random arbitrary test data it instead enumerates a deterministic series of test data to a fixed depth.

```
smallCheck :: Testable IO a => Depth -> a -> IO ()
list :: Depth -> Series Identity a -> [a]
sample' :: Gen a -> IO [a]
```

```
λ: list 3 series :: [Int]
[0,1,-1,2,-2,3,-3]

λ: list 3 series :: [Double]
[0.0,1.0,-1.0,2.0,0.5,-2.0,4.0,0.25,-0.5,-4.0,-0.25]

λ: list 3 series :: [(Int, String)]
[(0,""),(1,""),(0,"a"),(-1,""),(0,"b"),(1,"a"),(2,""),(1,"b"),(-1,"a"),(-2,""),(-1,"b"
```

It is useful to generate test cases over *all* possible inputs of a program up to some depth.

```
import Test.SmallCheck

distrib :: Int -> Int -> Int -> Bool
distrib a b c = a * (b + c) == a * b + a * c

cauchy :: [Double] -> [Double] -> Bool
cauchy xs ys = (abs (dot xs ys))^2 <= (dot xs xs) * (dot ys ys)

failure :: [Double] -> [Double] -> Bool
failure xs ys = abs (dot xs ys) <= (dot xs xs) * (dot ys ys)

dot :: Num a => [a] -> [a] -> a
dot xs ys = sum (zipWith (*) xs ys)

main :: IO ()
main = do
  putStrLn "Testing distributivity..."
  smallCheck 25 distrib

  putStrLn "Testing Cauchy-Schwarz..."
  smallCheck 4 cauchy

  putStrLn "Testing invalid Cauchy-Schwarz..."
  smallCheck 4 failure
```

```
$ runhaskell smallcheck.hs
Testing distributivity...
Completed 132651 tests without failure.

Testing Cauchy-Schwarz...
Completed 27556 tests without failure.

Testing invalid Cauchy-Schwarz...
Failed test no. 349.
there exist [1.0] [0.5] such that
  condition is false
```

Just like for QuickCheck we can implement series instances for our custom datatypes. For example there is no default instance for Vector, so let's implement one:

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}

import Test.SmallCheck
import Test.SmallCheck.Series
import Control.Applicative

import qualified Data.Vector as V

dot :: Num a => V.Vector a -> V.Vector a -> a
dot xs ys = V.sum (V.zipWith (*) xs ys)

cauchy :: V.Vector Double -> V.Vector Double -> Bool
cauchy xs ys = (abs (dot xs ys))^2 <= (dot xs xs) * (dot ys ys)

instance (Serial m a, Monad m) => Serial m (V.Vector a) where
  series = V.fromList <$> series

main :: IO ()
main = smallCheck 4 cauchy
```

SmallCheck can also use Generics to derive Serial instances, for example to enumerate all trees of a certain depth we might use:

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics
import Test.SmallCheck.Series

data Tree a = Null | Fork (Tree a) a (Tree a)
    deriving (Show, Generic)
```

```
instance Serial m a => Serial m (Tree a)

example :: [Tree ()]
example = list 3 series


main = print example
```

QuickSpec

Using the QuickCheck arbitrary machinery we can also rather remarkably enumerate a large number of combinations of functions to try and deduce algebraic laws from trying out inputs for small cases.

Of course the fundamental limitation of this approach is that a function may not exhibit any interesting properties for small cases or for simple function compositions. So in general case this approach won't work, but practically it still quite useful.

```
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE ScopedTypeVariables #-}

import Data.List
import Data.Typeable

import Test.QuickSpec hiding (lists, bools, arith)
import Test.QuickCheck

type Var k a = (Typeable a, Arbitrary a, CoArbitrary a, k a)

listCons :: forall a. Var Ord a => a -> Sig
listCons a = background
  [
    "[]"       `fun0` ([]        :: [a]),
    ":"        `fun2` ((:)       :: a -> [a] -> [a])
  ]

lists :: forall a. Var Ord a => a -> [Sig]
lists a =
  [
    -- Names to print arbitrary variables
    funs',
    funvars',
    vars',

    -- Ambient definitions
    listCons a,

    -- Expressions to deduce properties of
    "sort"     `fun1` (sort    :: [a] -> [a]),
    "map"      `fun2` (map     :: (a -> a) -> [a] -> [a]),
    "id"       `fun1` (id      :: [a] -> [a]),
    "reverse"  `fun1` (reverse :: [a] -> [a]),
```

```
        "minimum"  `fun1` (minimum :: [a] -> a),
        "length"   `fun1` (length  :: [a] -> Int),
        "++"       `fun2` ((++)    :: [a] -> [a] -> [a])
    ]

   where
     funs'    = funs (undefined :: a)
     funvars' = vars ["f", "g", "h"] (undefined :: a -> a)
     vars'    = ["xs", "ys", "zs"] `vars` (undefined :: [a])


tvar :: A
tvar = undefined

main :: IO ()
main = quickSpec (lists tvar)
```

Running this we rather see it is able to deduce most of the laws for list functions.

```
$ runhaskell src/quickspec.hs
== API ==
-- functions --
map :: (A -> A) -> [A] -> [A]
minimum :: [A] -> A
(++) :: [A] -> [A] -> [A]
length :: [A] -> Int
sort, id, reverse :: [A] -> [A]

-- background functions --
id :: A -> A
(:) :: A -> [A] -> [A]
(.) :: (A -> A) -> (A -> A) -> A -> A
[] :: [A]

-- variables --
f, g, h :: A -> A
xs, ys, zs :: [A]

-- the following types are using non-standard equality --
A -> A

-- WARNING: there are no variables of the following types; consider adding some --
A

== Testing ==
Depth 1: 12 terms, 4 tests, 24 evaluations, 12 classes, 0 raw equations.
Depth 2: 80 terms, 500 tests, 18673 evaluations, 52 classes, 28 raw equations.
Depth 3: 1553 terms, 500 tests, 255056 evaluations, 1234 classes, 319 raw equations.
319 raw equations; 1234 terms in universe.

== Equations about map ==
  1: map f [] == []
```

```
  2: map id xs == xs
  3: map (f.g) xs == map f (map g xs)

== Equations about minimum ==
  4: minimum [] == undefined

== Equations about (++) ==
  5: xs++[] == xs
  6: []++xs == xs
  7: (xs++ys)++zs == xs++(ys++zs)

== Equations about sort ==
  8: sort [] == []
  9: sort (sort xs) == sort xs

== Equations about id ==
 10: id xs == xs

== Equations about reverse ==
 11: reverse [] == []
 12: reverse (reverse xs) == xs

== Equations about several functions ==
 13: minimum (xs++ys) == minimum (ys++xs)
 14: length (map f xs) == length xs
 15: length (xs++ys) == length (ys++xs)
 16: sort (xs++ys) == sort (ys++xs)
 17: map f (reverse xs) == reverse (map f xs)
 18: minimum (sort xs) == minimum xs
 19: minimum (reverse xs) == minimum xs
 20: minimum (xs++xs) == minimum xs
 21: length (sort xs) == length xs
 22: length (reverse xs) == length xs
 23: sort (reverse xs) == sort xs
 24: map f xs++map f ys == map f (xs++ys)
 25: reverse xs++reverse ys == reverse (ys++xs)
```

Keep in mind the rather remarkable fact that this is all deduced automatically from the types alone!

Criterion

Criterion is a statistically aware benchmarking tool.

```
whnf :: (a -> b) -> a -> Pure
nf :: NFData b => (a -> b) -> a -> Pure
nfIO :: NFData a => IO a -> IO ()
bench :: Benchmarkable b => String -> b -> Benchmark
```

```
import Criterion.Main
```

```haskell
import Criterion.Config

-- Naive recursion for fibonacci numbers.
fib1 :: Int -> Int
fib1 0 = 0
fib1 1 = 1
fib1 n = fib1 (n-1) + fib1 (n-2)

-- Use the De Moivre closed form for fibonacci numbers.
fib2 :: Int -> Int
fib2 x = truncate $ ( 1 / sqrt 5 ) * ( phi ^ x - psi ^ x )
  where
      phi = ( 1 + sqrt 5 ) / 2
      psi = ( 1 - sqrt 5 ) / 2

suite :: [Benchmark]
suite = [
    bgroup "naive" [
      bench "fib 10" $ whnf fib1 5
    , bench "fib 20" $ whnf fib1 10
    ],
    bgroup "de moivre" [
      bench "fib 10" $ whnf fib2 5
    , bench "fib 20" $ whnf fib2 10
    ]
  ]

main :: IO ()
main = defaultMain suite
```

```
$ runhaskell criterion.hs
warming up
estimating clock resolution...
mean is 2.349801 us (320001 iterations)
found 1788 outliers among 319999 samples (0.6%)
  1373 (0.4%) high severe
estimating cost of a clock call...
mean is 65.52118 ns (23 iterations)
found 1 outliers among 23 samples (4.3%)
  1 (4.3%) high severe

benchmarking naive/fib 10
mean: 9.903067 us, lb 9.885143 us, ub 9.924404 us, ci 0.950
std dev: 100.4508 ns, lb 85.04638 ns, ub 123.1707 ns, ci 0.950

benchmarking naive/fib 20
mean: 120.7269 us, lb 120.5470 us, ub 120.9459 us, ci 0.950
std dev: 1.014556 us, lb 858.6037 ns, ub 1.296920 us, ci 0.950

benchmarking de moivre/fib 10
mean: 7.699219 us, lb 7.671107 us, ub 7.802116 us, ci 0.950
std dev: 247.3021 ns, lb 61.66586 ns, ub 572.1260 ns, ci 0.950
```

```
found 4 outliers among 100 samples (4.0%)
  2 (2.0%) high mild
  2 (2.0%) high severe
variance introduced by outliers: 27.726%
variance is moderately inflated by outliers

benchmarking de moivre/fib 20
mean: 8.082639 us, lb 8.018560 us, ub 8.350159 us, ci 0.950
std dev: 595.2161 ns, lb 77.46251 ns, ub 1.408784 us, ci 0.950
found 8 outliers among 100 samples (8.0%)
  4 (4.0%) high mild
  4 (4.0%) high severe
variance introduced by outliers: 67.628%
variance is severely inflated by outliers
```

Criterion can also generate a HTML page containing the benchmark results plotted

```
$ ghc -O2 --make criterion.hs
$ ./criterion -o bench.html
```


fib1

Mean execution time 138.5 µs   139.2 µs 140.8 µs
Standard deviation   2.870 µs   5.262 µs 9.244 µs

Outlying measurements have moderate (34.6%) effect on estimated standard deviation.
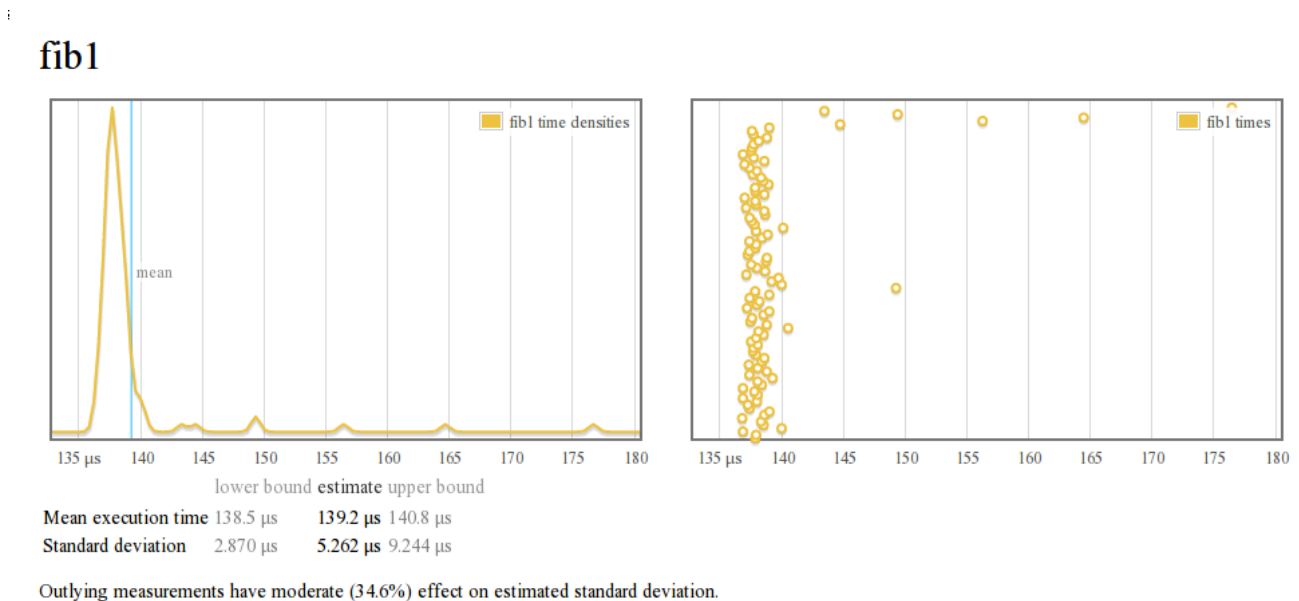
Tasty

Tasty combines all of the testing frameworks into a common API for forming runnable batches of tests and collecting the results.

```haskell
import Test.Tasty
import Test.Tasty.HUnit
import Test.Tasty.QuickCheck
import qualified Test.Tasty.SmallCheck as SC

arith :: Integer -> Integer -> Property
arith x y = (x > 0) && (y > 0) ==> (x+y)^2 > x^2 + y^2
```

```haskell
negation :: Integer -> Bool
negation x = abs (x^2) >= x

suite :: TestTree
suite = testGroup "Test Suite" [
    testGroup "Units"
      [ testCase "Equality" $ True @=? True
      , testCase "Assertion" $ assert $ (length [1,2,3]) == 3
      ],

    testGroup "QuickCheck tests"
      [ testProperty "Quickcheck test" arith
      ],

    testGroup "SmallCheck tests"
      [ SC.testProperty "Negation" negation
      ]
  ]

main :: IO ()
main = defaultMain suite
```

```
$ runhaskell TestSuite.hs
Unit tests
  Units
    Equality:          OK
    Assertion:         OK
  QuickCheck tests
    Quickcheck test: OK
      +++ OK, passed 100 tests.
  SmallCheck tests
    Negation:          OK
      11 tests completed
```

Type Families

MultiParam Typeclasses

Resolution of vanilla Haskell 98 typeclasses proceeds via very simple context reduction that minimizes interdependency between predicates, resolves superclasses, and reduces the types to head normal form. For example:

```haskell
(Eq [a], Ord [a]) => [a]
==> Ord a => [a]
```

If a single parameter typeclass expresses a property of a type (i.e. it's in a class or not in class) then a multiparameter typeclass expresses relationships between types. For example if we wanted to express the relation a type can be converted to another type we might use a class like:

```haskell
{-# LANGUAGE MultiParamTypeClasses #-}

import Data.Char

class Convertible a b where
  convert :: a -> b

instance Convertible Int Integer where
  convert = toInteger

instance Convertible Int Char where
  convert = chr

instance Convertible Char Int where
  convert = ord
```

Of course now our instances for `Convertible Int` are not unique anymore, so there no longer exists a nice procedure for determining the inferred type of `b` from just `a`. To remedy this let's add a functional dependency `a -> b`, which tells GHC that an instance `a` uniquely determines the instance that b can be. So we'll see that our two instances relating `Int` to both `Integer` and `Char` conflict.

```haskell
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}


import Data.Char

class Convertible a b | a -> b where
  convert :: a -> b

instance Convertible Int Char where
  convert = chr

instance Convertible Char Int where
  convert = ord
```

```
Functional dependencies conflict between instance declarations:
  instance Convertible Int Integer
  instance Convertible Int Char
```

Now there's a simpler procedure for determining instances uniquely and multiparameter typeclasses become more usable and inferable again. Effectively a functional dependency `| a -> b` says that we can't define multiple multiparamater typeclass instances with the same `a` but different `b`.

```
λ: convert (42 :: Int)
'42'
λ: convert '*'
```

Now let's make things not so simple. Turning on `UndecidableInstances` loosens the constraint on context reduction that can only allow constraints of the class to become structural smaller than its head. As a result implicit computation can now occur *within in the type class instance search*. Combined with a type-level representation of Peano numbers we find that we can encode basic arithmetic at the type-level.

```haskell
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE UndecidableInstances #-}

data Z
data S n

type Zero  = Z
type One   = S Zero
type Two   = S One
type Three = S Two
type Four  = S Three

zero :: Zero
zero = undefined

one :: One
one = undefined

two :: Two
two = undefined

three :: Three
three = undefined

four :: Four
four = undefined

class Eval a where
  eval :: a -> Int

instance Eval Zero where
  eval _ = 0

instance Eval n => Eval (S n) where
  eval m = 1 + eval (prev m)

class Pred a b | a -> b where
  prev :: a -> b

instance Pred Zero Zero where
  prev = undefined
```

```
instance Pred (S n) n where
  prev = undefined

class Add a b c | a b -> c where
  add :: a -> b -> c

instance Add Zero a a where
  add = undefined

instance Add a b c => Add (S a) b (S c) where
  add = undefined

f :: Three
f = add one two

g :: S (S (S (S Z)))
g = add two two

h :: Int
h = eval (add three four)
```

If the typeclass contexts look similar to Prolog you're not wrong, if one reads the contexts qualifier (=>) backwards as turnstiles (:-) then it's precisely the same equations.

```
add(0, A, A).
add(s(A), B, s(C)) :- add(A, B, C).

pred(0, 0).
pred(S(A), A).
```

This is kind of abusing typeclasses and if used carelessly it can fail to terminate or overflow at compile-time. UndecidableInstances shouldn't be turned on without careful forethought about what it implies.

```
<interactive>:1:1:
    Context reduction stack overflow; size = 201
```

Type Families

Type families allows us to write functions in the type domain which take types as arguments which can yield either types or values indexed on their arguments which are evaluated at compile-time in during typechecking. Type families come in two varieties: **data families** and **type synonym families**.

- **type families** are named function on types
- **data families** are type-indexed data types

First let's look at *type synonym families*, there are two equivalent syntactic ways of constructing them. Either as *associated* type families declared within a typeclass or as standalone declarations at the toplevel. The following forms are semantically equivalent, although the unassociated form is strictly more general:

```haskell
-- (1) Unassociated form
type family Rep a
type instance Rep Int = Char
type instance Rep Char = Int

class Convertible a where
  convert :: a -> Rep a

instance Convertible Int where
  convert = chr

instance Convertible Char where
  convert = ord



-- (2) Associated form
class Convertible a where
  type Rep a
  convert :: a -> Rep a

instance Convertible Int where
  type Rep Int = Char
  convert = chr

instance Convertible Char where
  type Rep Char = Int
  convert = ord
```

Using the same example we used for multiparameter + functional dependencies illustration we see that there is a direct translation between the type family approach and functional dependencies. These two approaches have the same expressive power.

An associated type family can be queried using the `:kind!` command in GHCi.

```
λ: :kind! Rep Int
Rep Int :: *
= Char
λ: :kind! Rep Char
Rep Char :: *
= Int
```

Data families on the other hand allow us to create new type parameterized data constructors. Normally we can only define typeclasses functions whose behavior results in a uniform result which is purely a result of the typeclasses arguments. With data families we can allow specialized behavior indexed on the type.

For example if we wanted to create more complicated vector structures (bit-masked vectors, vectors of tuples, ...) that exposed a uniform API but internally handled the differences in their data layout we can use data families to accomplish this:

```haskell
{-# LANGUAGE TypeFamilies #-}

import qualified Data.Vector.Unboxed as V

data family Array a
data instance Array Int      = IArray (V.Vector Int)
data instance Array Bool     = BArray (V.Vector Bool)
data instance Array (a,b)     = PArray (Array a) (Array b)
data instance Array (Maybe a) = MArray (V.Vector Bool) (Array a)

class IArray a where
  index :: Array a -> Int -> a

instance IArray Int where
  index (IArray xs) i = xs V.! i

instance IArray Bool where
  index (BArray xs) i = xs V.! i

-- Vector of pairs
instance (IArray a, IArray b) => IArray (a, b) where
  index (PArray xs ys) i = (index xs i, index ys i)

-- Vector of missing values
instance (IArray a) => IArray (Maybe a) where
  index (MArray bm xs) i =
    case bm V.! i of
      True  -> Nothing
      False -> Just $ index xs i
```

Injectivity

The type level functions defined by type-families are not necessarily *injective*, the function may map two distinct input types to the same output type. This differs from the behavior of type constructors (which are also type-level functions) which are injective.

For example for the constructor `Maybe`, `Maybe t1 = Maybe t2` implies that `t1 = t2`.

```haskell
data Maybe a = Nothing | Just a
-- Maybe a ~ Maybe b   implies   a ~ b

type instance F Int = Bool
type instance F Char = Bool

-- F a ~ F b does not imply   a ~ b, in general
```

Roles

Roles are a further level of specification for type variables parameters of datatypes.

- `nominal`

- (representational)
- (phantom)

They were added to the language to address a rather nasty and long-standing bug around the correspondence between a newtype and its runtime representation. The fundamental distinction that roles introduce is there are two notions of type equality:

- (nominal) - Two types are the same.
- (representational) - Two types have the same runtime representation.

```haskell
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype Age = MkAge { unAge :: Int }

type family Inspect x
type instance Inspect Age = Int
type instance Inspect Int = Bool

class Boom a where
  boom :: a -> Inspect a

instance Boom Int where
  boom = (== 0)

deriving instance Boom Age

-- GHC 7.6.3 exhibits undefined behavior
failure = boom (MkAge 3)
-- -6341068275333450897
```

Roles are normally inferred automatically, but with the (RoleAnnotations) extension they can be manually annotated. Except in rare cases this should not be necessary although it is helpful to know what is going on under the hood.

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE RoleAnnotations #-}

data Nat = Zero | Suc Nat

type role Vec nominal representational
data Vec :: Nat -> * -> * where
  Nil  :: Vec Zero a
  (:*) :: a -> Vec n a -> Vec (Suc n) a

type role App representational nominal
data App (f :: k -> *) (a :: k) = App (f a)
```

```
type role Mu nominal nominal
data Mu (f :: (k -> *) -> k -> *) (a :: k) = Roll (f (Mu f) a)

type role Proxy phantom
data Proxy (a :: k) = Proxy
```

See:

- [Roles: A New Feature of GHC](#)
- [Roles](#)

Monotraversable

Using type families, mono-traversable generalizes the notion of Functor, Foldable, and Traversable to include both monomorphic and polymorphic types.

```
omap :: MonoFunctor mono => (Element mono -> Element mono) -> mono -> mono

otraverse :: (Applicative f, MonoTraversable mono)
          => (Element mono -> f (Element mono)) -> mono -> f mono

ofoldMap :: (Monoid m, MonoFoldable mono)
         => (Element mono -> m) -> mono -> m
ofoldl' :: MonoFoldable mono
        => (a -> Element mono -> a) -> a -> mono -> a
ofoldr :: MonoFoldable mono
       => (Element mono -> b -> b) -> b -> mono -> b
```

For example the text type normally does not admit any of these type-classes since, but now we can write down the instances that model the interface of Foldable and Traversable.

```
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE OverloadedStrings #-}

import Data.Text
import Data.Char
import Data.Monoid
import Data.MonoTraversable
import Control.Applicative

bs :: Text
bs = "Hello Haskell."

shift :: Text
shift = omap (chr . (+1) . ord) bs
-- "Ifmmp!Ibtlfmm/"

backwards :: [Char]
backwards = ofoldl' (flip (:)) "" bs
-- ".lleksaH olleH"
```

```haskell
data MyMonoType = MNil | MCons Int MyMonoType deriving Show

type instance Element MyMonoType = Int

instance MonoFunctor MyMonoType where
  omap f MNil = MNil
  omap f (MCons x xs) = f x `MCons` omap f xs

instance MonoFoldable MyMonoType where
  ofoldMap f = ofoldr (mappend . f) mempty
  ofoldr       = mfoldr
  ofoldl'      = mfoldl'
  ofoldr1Ex f  = ofoldr1Ex f . mtoList
  ofoldl1Ex' f = ofoldl1Ex' f . mtoList

instance MonoTraversable MyMonoType where
  omapM f xs = mapM f (mtoList xs) >>= return . mfromList
  otraverse f = ofoldr acons (pure MNil)
    where acons x ys = MCons <$> f x <*> ys

mtoList :: MyMonoType -> [Int]
mtoList (MNil) = []
mtoList (MCons x xs) = x : (mtoList xs)

mfromList :: [Int] -> MyMonoType
mfromList [] = MNil
mfromList (x:xs) = MCons x (mfromList xs)

mfoldr :: (Int -> a -> a) -> a -> MyMonoType -> a
mfoldr f z MNil =  z
mfoldr f z (MCons x xs) =  f x (mfoldr f z xs)

mfoldl' :: (a -> Int -> a) -> a -> MyMonoType -> a
mfoldl' f z MNil = z
mfoldl' f z (MCons x xs) = let z' = z `f` x
                           in seq z' $ mfoldl' f z' xs

ex1 :: Int
ex1 = mfoldl' (+) 0 (mfromList [1..25])

ex2 :: MyMonoType
ex2 = omap (+1) (mfromList [1..25])
```

See: From Semigroups to Monads

NonEmpty

Rather than having degenerate (and often partial) cases of many of the Prelude functions to accommodate the null case of lists, it is sometimes preferable to statically enforce empty lists from even being constructed as an inhabitant of a type.

```haskell
infixr 5 :|, <|
data NonEmpty a = a :| [a]

head :: NonEmpty a -> a
toList :: NonEmpty a -> [a]
fromList :: [a] -> NonEmpty a
```

```haskell
head :: NonEmpty a -> a
head ~(a :| _) = a
```

```haskell
import Data.List.NonEmpty
import Prelude hiding (head, tail, foldl1)
import Data.Foldable (foldl1)

a :: NonEmpty Integer
a = fromList [1,2,3]
-- 1 :| [2,3]

b :: NonEmpty Integer
b = 1 :| [2,3]
-- 1 :| [2,3]

c :: NonEmpty Integer
c = fromList []
-- *** Exception: NonEmpty.fromList: empty list

d :: Integer
d = foldl1 (+) $ fromList [1..100]
-- 5050
```

In GHC 7.8 `-XOverloadedLists` can be used to avoid the extraneous `fromList` and `toList` conversions.

Manual Proofs

One of most deep results in computer science, the <u>Curry–Howard correspondence</u>, is the relation that logical propositions can be modeled by types and instantiating those types constitute proofs of these propositions. Programs are proofs and proofs are programs.

Types	Logic
`A`	proposition
`a : A`	proof
`B(x)`	predicate
`Void`	$\bot$
`Unit`	T
`A + B`	$A \lor B$
`A × B`	$A \land B$
`A -> B`	$A \Rightarrow B$

In dependently typed languages we can exploit this result to its full extent, in Haskell we don't have the strength that dependent types provide but can still prove trivial results. For example, now we can model a type level function for addition and provide a small proof that zero is an additive identity.

```
P 0                      [ base step ]
∀n. P n  → P (1+n)    [ inductive step ]
-------------------
∀n. P(n)
```

```
Axiom 1: a + 0 = a
Axiom 2: a + suc b = suc (a + b)

  0 + suc a
= suc (0 + a)  [by Axiom 2]
= suc a         [Induction hypothesis]
∎
```

Translated into Haskell our axioms are simply type definitions and recursing over the inductive datatype constitutes the inductive step of our proof.

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ExplicitForAll #-}
{-# LANGUAGE TypeOperators #-}

data Z
data S n

data SNat n where
  Zero :: SNat Z
  Succ :: SNat n -> SNat (S n)

data Eql a b where
  Refl :: Eql a a

type family Add m n
type instance Add Z n = n
type instance Add (S m) n = S (Add m n)

add :: SNat n -> SNat m -> SNat (Add n m)
add Zero      m = m
add (Succ n) m = Succ (add n m)

cong :: Eql a b -> Eql (f a) (f b)
cong Refl = Refl

-- ∀n. 0 + suc n = suc n
plus_suc :: forall n.  SNat n
         -> Eql (Add Z (S n)) (S n)
```

```
plus_suc Zero = Refl
plus_suc (Succ n) = cong (plus_suc n)

-- ∀n. 0 + n = n
plus_zero :: forall n. SNat n
          -> Eql (Add Z n) n
plus_zero Zero = Refl
plus_zero (Succ n) = cong (plus_zero n)
```

Using the `TypeOperators` extension we can also use infix notation at the type-level.

```
data a :=: b where
  Refl :: a :=: a

cong :: a :=: b -> (f a) :=: (f b)
cong Refl = Refl

type family (n :: Nat) :+ (m :: Nat) :: Nat
type instance Zero     :+ m = m
type instance (Succ n) :+ m = Succ (n :+ m)

plus_suc :: forall n m. SNat n -> SNat m -> (n :+ (S m)) :=: (S (n :+ m))
plus_suc Zero m = Refl
plus_suc (Succ n) m = cong (plus_suc n m)
```

Constraint Kinds

GHC's implementation also exposes the predicates that bound quantifiers in Haskell as types themselves, with the `-XConstraintKinds` extension enabled. Using this extension we work with constraints as first class types.

```
Num :: * -> Constraint
Odd :: * -> Constraint
```

```
type T1 a = (Num a, Ord a)
```

The empty constraint set is indicated by `() :: Constraint`.

For a contrived example if we wanted to create a generic `Sized` class that carried with it constraints on the elements of the container in question we could achieve this quite simply using type families.

```
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE ConstraintKinds #-}

import GHC.Exts (Constraint)
import Data.Hashable
```

```
import Data.HashSet

type family Con a :: Constraint
type instance Con [a] = (Ord a, Eq a)
type instance Con (HashSet a) = (Hashable a)

class Sized a where
  gsize :: Con a => a -> Int

instance Sized [a] where
  gsize = length

instance Sized (HashSet a) where
  gsize = size
```

One use-case of this is to capture the typeclass dictionary constrained by a function and reify it as a value.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE KindSignatures #-}

import GHC.Exts (Constraint)

data Dict :: Constraint -> * where
  Dict :: (c) => Dict c

dShow :: Dict (Show a) -> a -> String
dShow Dict x = show x

dEqNum :: Dict (Eq a, Num a) -> a -> Bool
dEqNum Dict x = x == 0


fShow :: String
fShow = dShow Dict 10

fEqual :: Bool
fEqual = dEqNum Dict 0
```

Both Constraints and AnyK types are somewhat unique in the Haskell implementation, in that they have the (BOX) kind.

```
λ: import GHC.Prim

λ: :kind AnyK
AnyK :: BOX

λ: :kind Constraint
Constraint :: BOX
```

Promotion

Higher Kinds

The kind system in Haskell is unique most other languages in that it allows datatypes to be constructed which take types and type constructor to other types. Such a system is said to support *higher kinded types*.

All kind annotations in Haskell necessarily result in a kind `*` although any terms to the left may be higher-kinded (`* -> *`).

The common example is the Monad which has kind `* -> *` . But we have also seen this higher-kindedness in free monads.

```haskell
data Free f a where
  Pure :: a -> Free f a
  Free :: f (Free f a) -> Free f a

data Cofree f a where
  Cofree :: a -> f (Cofree f a) -> Cofree f a
```

```haskell
Free   :: (* -> *) -> * -> *
Cofree :: (* -> *) -> * -> *
```

For instance `Cofree Maybe a` for some monokinded type `a` models a non-empty list with `Maybe :: * -> *` .

```haskell
-- Cofree Maybe a is a non-empty list
testCofree :: Cofree Maybe Int
testCofree = (Cofree 1 (Just (Cofree 2 Nothing)))
```

Kind Polymorphism

The regular value level function which takes a function and applies it to an argument is universally generalized over in the usual Hindley-Milner way.

```haskell
app :: forall a b. (a -> b) -> a -> b
app f a = f a
```

But when we do the same thing at the type-level we see we loose information about the polymorphism of the constructor applied.

```haskell
-- TApp :: (* -> *) -> * -> *
data TApp f a = MkTApp (f a)
```

Turning on `-XPolyKinds` allows polymorphic variables at the kind level as well.

```
-- Default:   (* -> *) -> * -> *
-- PolyKinds: (k -> *) -> k -> *
data TApp f a = MkTApp (f a)

-- Default:   ((* -> *) -> (* -> *)) -> (* -> *)
-- PolyKinds: ((k -> *) -> (k -> *)) -> (k -> *)
data Mu f a = Roll (f (Mu f) a)

-- Default:   * -> *
-- PolyKinds: k -> *
data Proxy a = Proxy
```

Using the polykinded `Proxy` type allows us to write down type class functions over constructors of arbitrary kind arity.

```
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE KindSignatures #-}

data Proxy a = Proxy
data Rep = Rep

class PolyClass a where
  foo :: Proxy a -> Rep
  foo = const Rep

-- () :: *
-- [] :: * -> *
-- Either :: * -> * -> *

instance PolyClass ()
instance PolyClass []
instance PolyClass Either
```

For example we can write down the polymorphic (S) (K) combinators at the type level now.

```
{-# LANGUAGE PolyKinds #-}

newtype I (a :: *) = I a
newtype K (a :: *) (b :: k) = K a
newtype Flip (f :: k1 -> k2 -> *) (x :: k2) (y :: k1) = Flip (f y x)

unI :: I a -> a
unI (I x) = x
```

```
unK :: K a b -> a
unK (K x) = x

unFlip :: Flip f x y -> f y x
unFlip (Flip x) = x
```

Data Kinds

The `-XDataKinds` extension allows us to use refer to constructors at the value level and the type level. Consider a simple sum type:

```
data S a b = L a | R b

-- S :: * -> * -> *
-- L :: a -> S a b
-- R :: b -> S a b
```

With the extension enabled we see that our type constructors are now automatically promoted so that Ⓛ or Ⓡ can be viewed as both a data constructor of the type Ⓢ or as the type Ⓛ with kind Ⓢ .

```
{-# LANGUAGE DataKinds #-}

data S a b = L a | R b

-- S :: * -> * -> *
-- L :: * -> S * *
-- R :: * -> S * *
```

Promoted data constructors can referred to in type signatures by prefixing them with a single quote. Also of importance is that these promoted constructors are not exported with a module by default, but type synonym instances can be created using this notation.

```
data Foo = Bar | Baz
type Bar = 'Bar
type Baz = 'Baz
```

Combining this with type families we see we can write meaningful, meaningful type-level functions by lifting types to the kind level.

```
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE DataKinds #-}

import Prelude hiding (Bool(..))
```

```
data Bool = True | False

type family Not (a :: Bool) :: Bool

type instance Not True  = False
type instance Not False = True

false :: Not True ~ False => a
false = undefined

true :: Not False ~ True => a
true = undefined

-- Fails at compile time.
-- Couldn't match type 'False with 'True
invalid :: Not True ~ True => a
invalid = undefined
```

Vectors

Using this new structure we can create a `Vec` type which is parameterized by its length as well as its element type now that we have a kind language rich enough to encode the successor type in the kind signature of the generalized algebraic datatype.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}

data Nat = Z | S Nat deriving (Eq, Show)

type Zero  = Z
type One   = S Zero
type Two   = S One
type Three = S Two
type Four  = S Three
type Five  = S Four

data Vec :: Nat -> * -> * where
  Nil  :: Vec Z a
  Cons :: a -> Vec n a -> Vec (S n) a

instance Show a => Show (Vec n a) where
  show Nil         = "Nil"
  show (Cons x xs) = "Cons " ++ show x ++ " (" ++ show xs ++ ")"

class FromList n where
  fromList :: [a] -> Vec n a

instance FromList Z where
  fromList [] = Nil
```

```haskell
instance FromList n => FromList (S n) where
  fromList (x:xs) = Cons x $ fromList xs


lengthVec :: Vec n a -> Nat
lengthVec Nil = Z
lengthVec (Cons x xs) = S (lengthVec xs)

zipVec :: Vec n a -> Vec n b -> Vec n (a,b)
zipVec Nil Nil = Nil
zipVec (Cons x xs) (Cons y ys) = Cons (x,y) (zipVec xs ys)

vec4 :: Vec Four Int
vec4 = fromList [0, 1, 2, 3]

vec5 :: Vec Five Int
vec5 = fromList [0, 1, 2, 3, 4]


example1 :: Nat
example1 = lengthVec vec4
-- S (S (S (S Z)))

example2 :: Vec Four (Int, Int)
example2 = zipVec vec4 vec4
-- Cons (0,0) (Cons (1,1) (Cons (2,2) (Cons (3,3) (Nil))))
```

So now if we try to zip two (Vec) types with the wrong shape then we get an error at compile-time about the off-by-one error.

```haskell
example2 = zipVec vec4 vec5
-- Couldn't match type 'S 'Z with 'Z
-- Expected type: Vec Four Int
--    Actual type: Vec Five Int
```

The same technique we can use to create a container which is statically indexed by an empty or non-empty flag, such that if we try to take the head of an empty list we'll get a compile-time error, or stated equivalently we have an obligation to prove to the compiler that the argument we hand to the head function is non-empty.

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}

data Size = Empty | NonEmpty

data List a b where
```

```
  Nil  :: List Empty a
  Cons :: a -> List b a -> List NonEmpty a

head' :: List NonEmpty a -> a
head' (Cons x _) = x

example1 :: Int
example1 = head' (1 `Cons` (2 `Cons` Nil))

-- Cannot match type Empty with NonEmpty
example2 :: Int
example2 = head' Nil
```

```
Couldn't match type None with Many
Expected type: List NonEmpty Int
  Actual type: List Empty Int
```

See:

- [Giving Haskell a Promotion](#)
- [Faking It: Simulating Dependent Types in Haskell](#)

Typelevel Numbers

GHC's type literals can also be used in place of explicit Peano arithmetic.

GHC 7.6 is very conservative about performing reduction, GHC 7.8 is much less so and will can solve many typelevel constraints involving natural numbers but sometimes still needs a little coaxing.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeOperators #-}

import GHC.TypeLits

data Vec :: Nat -> * -> * where
  Nil :: Vec 0 a
  Cons :: a -> Vec n a -> Vec (1 + n) a

-- GHC 7.6 will not reduce
-- vec3 :: Vec (1 + (1 + (1 + 0))) Int

vec3 :: Vec 3 Int
vec3 = 0 `Cons` (1 `Cons` (2 `Cons` Nil))
```

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
```

```haskell
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE FlexibleContexts #-}

import GHC.TypeLits
import Data.Type.Equality

data Foo :: Nat -> * where
  Small    :: (n <= 2)  => Foo n
  Big      :: (3 <= n) => Foo n

  Empty    :: ((n == 0) ~ True) => Foo n
  NonEmpty :: ((n == 0) ~ False) => Foo n

big :: Foo 10
big = Big

small :: Foo 2
small = Small

empty :: Foo 0
empty = Empty

nonempty :: Foo 3
nonempty = NonEmpty
```

See: Type-Level Literals

Type Equality

Continuing with the theme of building more elaborate proofs in Haskell, GHC 7.8 recently shipped with the `Data.Type.Equality` module which provides us with an extended set of type-level operations for expressing the equality of types as values, constraints, and promoted booleans.

```haskell
(~)   :: k -> k -> Constraint
(==)  :: k -> k -> Bool
(<=)  :: Nat -> Nat -> Constraint
(<=?) :: Nat -> Nat -> Bool
(+)   :: Nat -> Nat -> Nat
(-)   :: Nat -> Nat -> Nat
(*)   :: Nat -> Nat -> Nat
(^)   :: Nat -> Nat -> Nat
```

```haskell
(:~:)      :: k -> k -> *
Refl       :: a1 :~: a1
sym        :: (a :~: b) -> b :~: a
trans      :: (a :~: b) -> (b :~: c) -> a :~: c
castWith   :: (a :~: b) -> a -> b
gcastWith  :: (a :~: b) -> (a ~ b => r) -> r
```

With this we have a much stronger language for writing restrictions that can be checked at a compile-time, and a mechanism that will later allow us to write more advanced proofs.

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE ConstraintKinds #-}

import GHC.TypeLits
import Data.Type.Equality

type Not a b = ((b == a) ~ False)

restrictUnit :: Not () a => a -> a
restrictUnit = id

restrictChar :: Not Char a => a -> a
restrictChar = id
```

Proxy

Using kind polymorphism with phantom types allows us to express the Proxy type which is inhabited by a single constructor with no arguments but with a polykinded phantom type variable which carries an arbitrary type as the value is passed around.

```haskell
{-# LANGUAGE PolyKinds #-}

-- | A concrete, poly-kinded proxy type
data Proxy t = Proxy
```

```haskell
import Data.Proxy

a :: Proxy ()
a = Proxy

b :: Proxy 3
b = Proxy

c :: Proxy "symbol"
c = Proxy

d :: Proxy Maybe
d = Proxy

e :: Proxy (Maybe ())
e = Proxy
```

This is provided by the Prelude in 7.8.

Promoted Syntax

We've seen constructors promoted using DataKinds, but just like at the value-level GHC also allows us some syntactic sugar for list and tuples instead of explicit cons'ing and pair'ing. This is enabled with the `-XTypeOperators` extension, which introduces list syntax and tuples of arbitrary arity at the type-level.

```haskell
data HList :: [*] -> * where
  HNil  :: HList '[]
  HCons :: a -> HList t -> HList (a ': t)

data Tuple :: (*,*) -> * where
  Tuple :: a -> b -> Tuple '(a,b)
```

Using this we can construct all variety of composite type-level objects.

```haskell
λ: :kind 1
1 :: Nat

λ: :kind "foo"
"foo" :: Symbol

λ: :kind [1,2,3]
[1,2,3] :: [Nat]

λ: :kind [Int, Bool, Char]
[Int, Bool, Char] :: [*]

λ: :kind Just [Int, Bool, Char]
Just [Int, Bool, Char] :: Maybe [*]

λ: :kind '("a", Int)
(,) Symbol *

λ: :kind [ '("a", Int), '("b", Bool) ]
[ '("a", Int), '("b", Bool) ] :: [(,) Symbol *]
```

Singleton Types

A singleton type is a type with a single value inhabitant. Singleton types can be constructed in a variety of ways using GADTs or with data families.

```haskell
data instance Sing (a :: Nat) where
  SZ :: Sing 'Z
  SS :: Sing n -> Sing ('S n)

data instance Sing (a :: Maybe k) where
  SNothing :: Sing 'Nothing
  SJust :: Sing x -> Sing ('Just x)
```

```haskell
data instance Sing (a :: Bool) where
  STrue :: Sing True
  SFalse :: Sing False
```

Promoted Naturals

```
Value-level  Type-level        Models
-----------  -----------       -------
SZ           Sing 'Z           0
SS SZ        Sing ('S 'Z)      1
SS (SS SZ)   Sing ('S ('S 'Z)) 2
```

Promoted Booleans

```
Value-level  Type-level       Models
-----------  --------------   -------
STrue        Sing 'False      False
SFalse       Sing 'True       True
```

Promoted Maybe

```
Value-level  Type-level       Models
-----------  --------------   -------
SJust a      Sing (SJust 'a)  Just a
SNothing     Sing Nothing     Nothing
```

Singleton types are an integral part of the small cottage industry of faking dependent types in Haskell, i.e. constructing types with terms predicated upon values. Singleton types are a way of "cheating" by modeling the map between types and values as a structural property of the type.

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE TypeSynonymInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE UndecidableInstances #-}

import Data.Proxy
```

```haskell
import GHC.Exts (Any)
import Prelude hiding (succ)

data Nat = Z | S Nat

-- kind-indexed data family
data family Sing (a :: k)

data instance Sing (a :: Nat) where
  SZ :: Sing 'Z
  SS :: Sing n -> Sing ('S n)

data instance Sing (a :: Maybe k) where
  SNothing :: Sing 'Nothing
  SJust :: Sing x -> Sing ('Just x)

data instance Sing (a :: Bool) where
  STrue :: Sing True
  SFalse :: Sing False

data Fin (n :: Nat) where
  FZ :: Fin (S n)
  FS :: Fin n -> Fin (S n)

data Vec a n where
  Nil  :: Vec a Z
  Cons :: a -> Vec a n -> Vec a (S n)

class SingI (a :: k) where
  sing :: Sing a

instance SingI Z where
  sing = SZ

instance SingI n => SingI (S n) where
  sing = SS sing

deriving instance Show Nat
deriving instance Show (SNat a)
deriving instance Show (SBool a)
deriving instance Show (Fin a)
deriving instance Show a => Show (Vec a n)

type family (m :: Nat) :+ (n :: Nat) :: Nat where
  Z :+ n = n
  S m :+ n = S (m :+ n)

type SNat (k :: Nat) = Sing k
type SBool (k :: Bool) = Sing k
type SMaybe (b :: a) (k :: Maybe a) = Sing k

size :: Vec a n -> SNat n
size Nil         = SZ
size (Cons x xs) = SS (size xs)
```

```haskell
forget :: SNat n -> Nat
forget SZ = Z
forget (SS n) = S (forget n)

natToInt :: Integral n => Nat -> n
natToInt Z     = 0
natToInt (S n) = natToInt n + 1

intToNat :: (Integral a, Ord a) => a -> Nat
intToNat 0 = Z
intToNat n = S $ intToNat (n - 1)

sNatToInt :: Num n => SNat x -> n
sNatToInt SZ     = 0
sNatToInt (SS n) = sNatToInt n + 1

index :: Fin n -> Vec a n -> a
index FZ (Cons x _)      = x
index (FS n) (Cons _ xs) = index n xs


test1 :: Fin (S (S (S Z)))
test1 = FS (FS FZ)

test2 :: Int
test2 = index FZ (1 `Cons` (2 `Cons` Nil))

test3 :: Sing ('Just ('S ('S Z)))
test3 = SJust (SS (SS SZ))

test4 :: Sing ('S ('S Z))
test4 = SS (SS SZ)

-- polymorphic constructor SingI
test5 :: Sing ('S ('S Z))
test5 = sing
```

The builtin singleton types provided in `GHC.TypeLits` have the useful implementation that type-level values can be reflected to the value-level and back up to the type-level, albeit under an existential.

```haskell
someNatVal :: Integer -> Maybe SomeNat
someSymbolVal :: String -> SomeSymbol

natVal :: KnownNat n => proxy n -> Integer
symbolVal :: KnownSymbol n => proxy n -> String
```

```haskell
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
```

```
import Data.Proxy
import GHC.TypeLits

a :: Integer
a = natVal (Proxy :: Proxy 1)
-- 1

b :: String
b = symbolVal (Proxy :: Proxy "foo")
-- "foo"

c :: Integer
c = natVal (Proxy :: Proxy (2 + 3))
-- 5
```

Closed Type Families

In the type families we've used so far (called open type families) there is no notion of ordering of the equations involved in the type-level function. The type family can be extended at any point in the code resolution simply proceeds sequentially through the available definitions. Closed type-families allow an alternative declaration that allows for a base case for the resolution allowing us to actually write recursive functions over types.

For example consider if we wanted to write a function which counts the arguments in the type of a function and reifies at the value-level.

```
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}

import Data.Proxy
import GHC.TypeLits

type family Count (f :: *) :: Nat where
  Count (a -> b) = 1 + (Count b)
  Count x = 1

type Fn1 = Int -> Int
type Fn2 = Int -> Int -> Int -> Int

fn1 :: Integer
fn1 = natVal (Proxy :: Proxy (Count Fn1))
-- 2

fn2 :: Integer
fn2 = natVal (Proxy :: Proxy (Count Fn2))
-- 4
```

The variety of functions we can now write down are rather remarkable, allowing us to write meaningful logic at the type level.

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE UndecidableInstances #-}

import GHC.TypeLits
import Data.Proxy
import Data.Type.Equality

-- Type-level functions over type-level lists.

type family Reverse (xs :: [k]) :: [k] where
  Reverse '[] = '[]
  Reverse xs = Rev xs '[]

type family Rev (xs :: [k]) (ys :: [k]) :: [k] where
  Rev '[] i = i
  Rev (x ': xs) i = Rev xs (x ': i)

type family Length (as :: [k]) :: Nat where
  Length '[] = 0
  Length (x ': xs) = 1 + Length xs

type family If (p :: Bool) (a :: k) (b :: k) :: k where
  If True a b = a
  If False a b = b

type family Concat (as :: [k]) (bs :: [k]) :: [k] where
  Concat a '[] = a
  Concat '[] b = b
  Concat (a ': as) bs = a ': Concat as bs

type family Map (f :: a -> b) (as :: [a]) :: [b] where
  Map f '[] = '[]
  Map f (x ': xs) = f x ': Map f xs

type family Sum (xs :: [Nat]) :: Nat where
  Sum '[] = 0
  Sum (x ': xs) = x + Sum xs

ex1 :: Reverse [1,2,3] ~ [3,2,1] => Proxy a
ex1 = Proxy

ex2 :: Length [1,2,3] ~ 3 => Proxy a
ex2 = Proxy

ex3 :: (Length [1,2,3]) ~ (Length (Reverse [1,2,3])) => Proxy a
ex3 = Proxy

-- Reflecting type level computations back to the value level.
```

```haskell
ex4 :: Integer
ex4 = natVal (Proxy :: Proxy (Length (Concat [1,2,3] [4,5,6])))
-- 6

ex5 :: Integer
ex5 = natVal (Proxy :: Proxy (Sum [1,2,3]))
-- 6

-- Couldn't match type '2' with '1'
ex6 :: Reverse [1,2,3] ~ [3,1,2] => Proxy a
ex6 = Proxy
```

The results of type family functions need not necessarily be kinded as (*) either. For example using Nat or Constraint is permitted.

```haskell
type family Elem (a :: k) (bs :: [k]) :: Constraint where
  Elem a (a ': bs) = (() :: Constraint)
  Elem a (b ': bs) = a `Elem` bs

type family Sum (ns :: [Nat]) :: Nat where
  Sum '[] = 0
  Sum (n ': ns) = n + Sum ns
```

Kind Indexed Type Families

Just as typeclasses are normally indexed on types, classes can also be indexed on kinds with the kinds given as explicit kind signatures on type variables.

```haskell
type family (a :: k) == (b :: k) :: Bool
type instance a == b = EqStar a b
type instance a == b = EqArrow a b
type instance a == b = EqBool a b

type family EqStar (a :: *) (b :: *) where
  EqStar a a = True
  EqStar a b = False

type family EqArrow (a :: k1 -> k2) (b :: k1 -> k2) where
  EqArrow a a = True
  EqArrow a b = False

type family EqBool a b where
  EqBool True  True  = True
  EqBool False False = True
  EqBool a     b     = False

type family EqList a b where
  EqList '[]          '[]          = True
  EqList (h1 ': t1) (h2 ': t2) = (h1 == h2) && (t1 == t2)
```

```
  EqList a         b          = False
```

Promoted Symbols

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE ConstraintKinds #-}


import GHC.TypeLits
import Data.Type.Equality

data Label (l :: Symbol) = Get

class Has a l b | a l -> b where
  from :: a -> Label l -> b

data Point2D = Point2 Double Double deriving Show
data Point3D = Point3 Double Double Double deriving Show

instance Has Point2D "x" Double where
  from (Point2 x _) _ = x

instance Has Point2D "y" Double where
  from (Point2 _ y) _ = y


instance Has Point3D "x" Double where
  from (Point3 x _ _) _ = x

instance Has Point3D "y" Double where
  from (Point3 _ y _) _ = y

instance Has Point3D "z" Double where
  from (Point3 _ _ z) _ = z


infixl 6 #

(#) :: a -> (a -> b) -> b
(#) = flip ($)

_x :: Has a "x" b => a -> b
_x pnt = from pnt (Get :: Label "x")


_y :: Has a "y" b => a -> b
_y pnt = from pnt (Get :: Label "y")
```

```
_z :: Has a "z" b => a -> b
_z pnt = from pnt (Get :: Label "z")

type Point a r = (Has a "x" r, Has a "y" r)

distance :: (Point a r, Point b r, Floating r) => a -> b -> r
distance p1 p2 = sqrt (d1^2 + d2^2)
  where
    d1 = (p1 # _x) + (p1 # _y)
    d2 = (p2 # _x) + (p2 # _y)

main :: IO ()
main = do
  print $ (Point2 10 20) # _x

  -- Fails with: No instance for (Has Point2D "z" a0)
  -- print $ (Point2 10 20) # _z

  print $ (Point3 10 20 30) # _x
  print $ (Point3 10 20 30) # _z

  print $ distance (Point2 1 3) (Point2 2 7)
  print $ distance (Point2 1 3) (Point3 2 7 4)
  print $ distance (Point3 1 3 5) (Point3 2 7 3)
```

Since record is fundamentally no different from the tuple we can also do the same kind of construction over record field names.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE ExistentialQuantification #-}
{-# LANGUAGE ConstraintKinds #-}


import GHC.TypeLits

newtype Field (n :: Symbol) v = Field { unField :: v }
  deriving Show

data Person1 = Person1
  { _age     :: Field "age" Int
  , _name    :: Field "name" String
  }

data Person2 = Person2
  { _age'   :: Field "age" Int
```

```
    , _name' :: Field "name" String
    , _lib'  :: Field "lib" String
    }

deriving instance Show Person1
deriving instance Show Person2

data Label (l :: Symbol) = Get

class Has a l b | a l -> b where
  from :: a -> Label l -> b

instance Has Person1 "age" Int where
  from (Person1 a _) _ = unField a

instance Has Person1 "name" String where
  from (Person1 _ a) _ = unField a

instance Has Person2 "age" Int where
  from (Person2 a _ _) _ = unField a

instance Has Person2 "name" String where
  from (Person2 _ a _) _ = unField a

age :: Has a "age" b => a -> b
age pnt = from pnt (Get :: Label "age")

name :: Has a "name" b => a -> b
name pnt = from pnt (Get :: Label "name")

-- Parameterized constraint kind for "Simon-ness" of a record.
type Simon a = (Has a "name" String, Has a "age" Int)

spj :: Person1
spj = Person1 (Field 56) (Field "Simon Peyton Jones")

smarlow :: Person2
smarlow = Person2 (Field 38) (Field "Simon Marlow") (Field "rts")


catNames :: (Simon a, Simon b) => a -> b -> String
catNames a b = name a ++ name b

addAges :: (Simon a, Simon b) => a -> b -> Int
addAges a b = age a + age b


names :: String
names = name smarlow ++ "," ++ name spj
-- "Simon Marlow,Simon Peyton Jones"

ages :: Int
ages = age spj + age smarlow
-- 94
```

Notably this approach is mostly just all boilerplate class instantiation which could be abstracted away using TemplateHaskell or a Generic deriving.

HLists

A heterogeneous list is a cons list whose type statically encodes the ordered types of its values.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE KindSignatures #-}

infixr 5 :::

data HList (ts :: [ * ]) where
  Nil :: HList '[]
  (:::) :: t -> HList ts -> HList (t ': ts)

-- Take the head of a non-empty list with the first value as Bool type.
headBool :: HList (Bool ': xs) -> Bool
headBool hlist = case hlist of
  (a ::: _) -> a

hlength :: HList x -> Int
hlength Nil = 0
hlength (_ ::: b) = 1 + (hlength b)


tuple :: (Bool, (String, (Double, ())))
tuple = (True, ("foo", (3.14, ())))

hlist :: HList '[Bool, String , Double , ()]
hlist = True ::: "foo" ::: 3.14 ::: () ::: Nil
```

Of course this immediately begs the question of how to print such a list out to a string in the presence of type-heterogeneity. In this case we can use type-families combined with constraint kinds to apply the Show over the HLists parameters to generate the aggregate constraint that all types in the HList are Showable, and then derive the Show instance.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE UndecidableInstances #-}

import GHC.Exts (Constraint)
```

```haskell
infixr 5 :::

data HList (ts :: [ * ]) where
  Nil :: HList '[]
  (:::) :: t -> HList ts -> HList (t ': ts)

type family Map (f :: a -> b) (xs :: [a]) :: [b]
type instance Map f '[] = '[]
type instance Map f (x ': xs) = f x ': Map f xs

type family Constraints (cs :: [Constraint]) :: Constraint
type instance Constraints '[] = ()
type instance Constraints (c ': cs) = (c, Constraints cs)

type AllHave (c :: k -> Constraint) (xs :: [k]) = Constraints (Map c xs)

showHList :: AllHave Show xs => HList xs -> [String]
showHList Nil = []
showHList (x ::: xs) = (show x) : showHList xs

instance AllHave Show xs => Show (HList xs) where
  show = show . showHList

example1 :: HList '[Bool, String , Double , ()]
example1 = True ::: "foo" ::: 3.14 ::: () ::: Nil
-- ["True","\"foo\"","3.14","()"]
```

Typelevel Maps

Much of this discussion of promotion begs the question whether we can create data structures at the type-level to store information at compile-time. For example a type-level association list can be used to model a map between type-level symbols and any other promotable types. Together with type-families we can write down type-level traversal and lookup functions.

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE ConstraintKinds #-}
{-# LANGUAGE UndecidableInstances #-}

import GHC.TypeLits
import Data.Proxy
import Data.Type.Equality

type family If (p :: Bool) (a :: k) (b :: k) :: k where
  If True a b = a
  If False a b = b
```

```
type family Lookup (k :: a) (ls :: [(a, b)]) :: Maybe b where
  Lookup k '[] = 'Nothing
  Lookup k ('(a, b) ': xs) = If (a == k) ('Just b) (Lookup k xs)

type M = [
    '("a", 1)
  , '("b", 2)
  , '("c", 3)
  , '("d", 4)
  ]

type K = "a"
type (!!) m (k :: Symbol) a = (Lookup k m) ~ Just a

value :: Integer
value = natVal ( Proxy :: (M !! "a") a => Proxy a )
```

If we ask GHC to expand out the type signature we can view the explicit implementation of the type-level map lookup function.

```
(!!)
  :: If
       (GHC.TypeLits.EqSymbol "a" k)
       ('Just 1)
       (If
          (GHC.TypeLits.EqSymbol "b" k)
          ('Just 2)
          (If
             (GHC.TypeLits.EqSymbol "c" k)
             ('Just 3)
             (If (GHC.TypeLits.EqSymbol "d" k) ('Just 4) 'Nothing)))
     ~ 'Just v =>
     Proxy k -> Proxy v
```

Advanced Proofs

Now that we have the length-indexed vector let's go write the reverse function, how hard could it be?

So we go and write down something like this:

```
reverseNaive :: forall n a. Vec a n -> Vec a n
reverseNaive xs = go Nil xs -- Error: n + 0 != n
  where
    go :: Vec a m -> Vec a n -> Vec a (n :+ m)
    go acc Nil = acc
    go acc (Cons x xs) = go (Cons x acc) xs -- Error: n + succ m != succ (n + m)
```

Running this we find that GHC is unhappy about two lines in the code:

```
Couldn't match type 'n' with 'n :+ 'Z'
    Expected type: Vec a n
      Actual type: Vec a (n :+ 'Z)

Could not deduce ((n1 :+ 'S m) ~ 'S (n1 :+ m))
    Expected type: Vec a1 (k :+ m)
      Actual type: Vec a1 (n1 :+ 'S m)
```

As we unfold elements out of the vector we'll end up a doing a lot of type-level arithmetic over indices as we combine the subparts of the vector backwards, but as a consequence we find that GHC will run into some unification errors because it doesn't know about basic arithmetic properties of the natural numbers. Namely that `forall n. n + 0 = 0` and `forall n m. n + (1 + m) = 1 + (n + m)`. Which of course it really shouldn't be given that we've constructed a system at the type-level which intuitively *models* arithmetic but GHC is just a dumb compiler, it can't automatically deduce the isomorphism between natural numbers and Peano numbers.

So at each of these call sites we now have a proof obligation to construct proof terms which rearrange the type signatures of the terms in question such that actual types in the error messages GHC gave us align with the expected values to complete the program.

Recall from our discussion of propositional equality from GADTs that we actually have such machinery to do this!

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE ExplicitForAll #-}

import Data.Type.Equality

data Nat = Z | S Nat

data SNat n where
  Zero :: SNat Z
  Succ :: SNat n -> SNat (S n)

data Vec :: * -> Nat -> * where
  Nil :: Vec a Z
  Cons :: a -> Vec a n -> Vec a (S n)

instance Show a => Show (Vec a n) where
  show Nil         = "Nil"
  show (Cons x xs) = "Cons " ++ show x ++ " (" ++ show xs ++ ")"

type family (m :: Nat) :+ (n :: Nat) :: Nat where
  Z :+ n = n
  S m :+ n = S (m :+ n)

-- (a ~ b) implies (f a ~ f b)
```

```
cong :: a :~: b -> f a :~: f b
cong Refl = Refl

-- (a ~ b) implies (f a) implies (f b)
subst :: a :~: b -> f a -> f b
subst Refl = id

plus_zero :: forall n. SNat n -> (n :+ Z) :~: n
plus_zero Zero = Refl
plus_zero (Succ n) = cong (plus_zero n)

plus_suc :: forall n m. SNat n -> SNat m -> (n :+ (S m)) :~: (S (n :+ m))
plus_suc Zero m = Refl
plus_suc (Succ n) m = cong (plus_suc n m)

size :: Vec a n -> SNat n
size Nil         = Zero
size (Cons _ xs) = Succ $ size xs

reverse :: forall n a. Vec a n -> Vec a n
reverse xs = subst (plus_zero (size xs)) $ go Nil xs
  where
    go :: Vec a m -> Vec a k -> Vec a (k :+ m)
    go acc Nil = acc
    go acc (Cons x xs) = subst (plus_suc (size xs) (size acc)) $ go (Cons x acc) xs

append :: Vec a n -> Vec a m -> Vec a (n :+ m)
append (Cons x xs) ys = Cons x (append xs ys)
append Nil         ys = ys

vec :: Vec Int (S (S (S Z)))
vec = 1 `Cons` (2 `Cons` (3 `Cons` Nil))

test :: Vec Int (S (S (S Z)))
test = Main.reverse vec
```

One might consider whether we could avoid using the singleton trick and just use type-level natural numbers, and technically this approach should be feasible although it seems that the natural number solver in GHC 7.8 can decide some properties but not the ones needed to complete the natural number proofs for the reverse functions.

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE ExplicitForAll #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE UndecidableInstances #-}

import Prelude hiding (Eq)
import GHC.TypeLits
import Data.Type.Equality

type Z = 0
```

```haskell
type family S (n :: Nat) :: Nat where
  S n = n + 1

-- Yes!
eq_zero :: Z :~: Z
eq_zero = Refl

-- Yes!
zero_plus_one :: (Z + 1) :~: (1 + Z)
zero_plus_one = Refl

-- Yes!
plus_zero :: forall n. (n + Z) :~: n
plus_zero = Refl

-- Yes!
plus_one :: forall n. (n + S Z) :~: S n
plus_one = Refl

-- No.
plus_suc :: forall n m. (n + (S m)) :~: (S (n + m))
plus_suc = Refl
```

Caveat should be that there might be a way to do this in GHC 7.6 that I'm not aware of. In GHC 7.10 there are some planned changes to solver that should be able to resolve these issues. In particular there are plans to allow pluggable type system extensions that could outsource these kind of problems to third party SMT solvers which can solve these kind of numeric relations and return this information back to GHC's typechecker.

As an aside this is a direct transliteration of the equivalent proof in Agda, which is accomplished via the same method but without the song and dance to get around the lack of dependent types.

```agda
module Vector where

infixr 10 _::_

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

{-# BUILTIN NATURAL ℕ    #-}
{-# BUILTIN ZERO    zero #-}
{-# BUILTIN SUC     suc  #-}

infixl 6 _+_

_+_ : ℕ → ℕ → ℕ
0 + n = n
suc m + n = suc (m + n)

data Vec (A : Set) : ℕ → Set where
  []  : Vec A 0
```

```
  _::_ : ∀ {n} → A → Vec A n → Vec A (suc n)

_++_ : ∀ {A n m} → Vec A n → Vec A m → Vec A (n + m)
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

infix 4 _≡_

data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x

subst : {A : Set} → (P : A → Set) → ∀{x y} → x ≡ y → P x → P y
subst P refl p = p

cong : {A B : Set} (f : A → B) → {x y : A} → x ≡ y → f x ≡ f y
cong f refl = refl

vec : ∀ {A} (k : ℕ) → Set
vec {A} k = Vec A k

plus_zero : {n : ℕ} → n + 0 ≡ n
plus_zero {zero}  = refl
plus_zero {suc n} = cong suc plus_zero

plus_suc : {n : ℕ} → n + (suc 0) ≡ suc n
plus_suc {zero}  = refl
plus_suc {suc n} = cong suc (plus_suc {n})

reverse : ∀ {A n} → Vec A n → Vec A n
reverse []        = []
reverse {A} {suc n} (x :: xs) = subst vec (plus_suc {n}) (reverse xs ++ (x :: []))
```

Generics

Haskell has several techniques for automatic generation of type classes for a variety of tasks that
consist largely of boilerplate code generation such as:

- Pretty Printing
- Equality
- Serialization
- Ordering
- Traversal

Typeable

The `Typeable` class be used to create runtime type information for arbitrary types.

```
typeOf :: Typeable a => a -> TypeRep
```

```
{-# LANGUAGE DeriveDataTypeable #-}
```

```
import Data.Typeable

data Animal = Cat | Dog deriving Typeable
data Zoo a = Zoo [a] deriving Typeable

equal :: (Typeable a, Typeable b) => a -> b -> Bool
equal a b = typeOf a == typeOf b

example1 :: TypeRep
example1 = typeOf Cat
-- Animal

example2 :: TypeRep
example2 = typeOf (Zoo [Cat, Dog])
-- Zoo Animal

example3 :: TypeRep
example3 = typeOf ((1, 6.636e-34, "foo") :: (Int, Double, String))
-- (Int,Double,[Char])

example4 :: Bool
example4 = equal False ()
-- False
```

Using the Typeable instance allows us to write down a type safe cast function which can safely use `unsafeCast` and provide a proof that the resulting type matches the input.

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
cast x
  | typeOf x == typeOf ret = Just ret
  | otherwise = Nothing
  where
    ret = unsafeCast x
```

Of historical note is that writing our own Typeable classes is currently possible of GHC 7.6 but allows us to introduce dangerous behavior that can cause crashes, and shouldn't be done except by GHC itself. As of 7.8 GHC forbids hand-written Typeable instances.

See: Typeable and Data in Haskell

Dynamic

Since we have a way of querying runtime type information we can use this machinery to implement a `Dynamic` type. This allows us to box up any monotype into a uniform type that can be passed to any function taking a Dynamic type which can then unpack the underlying value in a type-safe way.

```
toDyn :: Typeable a => a -> Dynamic
fromDyn :: Typeable a => Dynamic -> a -> a
fromDynamic :: Typeable a => Dynamic -> Maybe a
cast :: (Typeable a, Typeable b) => a -> Maybe b
```

```haskell
import Data.Dynamic
import Data.Maybe

dynamicBox :: Dynamic
dynamicBox = toDyn (6.62 :: Double)

example1 :: Maybe Int
example1 = fromDynamic dynamicBox
-- Nothing

example2 :: Maybe Double
example2 = fromDynamic dynamicBox
-- Just 6.62

example3 :: Int
example3 = fromDyn dynamicBox 0
-- 0

example4 :: Double
example4 = fromDyn dynamicBox 0.0
-- 6.62
```

In GHC 7.8 the Typeable class is poly-kinded so polymorphic functions can be applied over dynamic objects.

Data

Just as Typeable let's create runtime type information where needed, the Data class allows us to reflect information about the structure of datatypes to runtime as needed.

```haskell
class Typeable a => Data a where
  gfoldl  :: (forall d b. Data d => c (d -> b) -> d -> c b)
          -> (forall g. g -> c g)
          -> a
          -> c a

  gunfold :: (forall b r. Data b => c (b -> r) -> c r)
          -> (forall r. r -> c r)
          -> Constr
          -> c a

  toConstr :: a -> Constr
  dataTypeOf :: a -> DataType
  gmapQl :: (r -> r' -> r) -> r -> (forall d. Data d => d -> r') -> a -> r
```

The types for `gfoldl` and `gunfold` are a little intimidating (and depend on `Rank2Types`), the best way to understand is to look at some examples. First the most trivial case a simple sum type `Animal` would produce the following code:

```haskell
data Animal = Cat | Dog deriving Typeable


instance Data Animal where
  gfoldl k z Cat = z Cat
  gfoldl k z Dog = z Dog

  gunfold k z c
    = case constrIndex c of
        1 -> z Cat
        2 -> z Dog

  toConstr Cat = cCat
  toConstr Dog = cDog

  dataTypeOf _ = tAnimal

tAnimal :: DataType
tAnimal = mkDataType "Main.Animal" [cCat, cDog]

cCat :: Constr
cCat = mkConstr tAnimal "Cat" [] Prefix

cDog :: Constr
cDog = mkConstr tAnimal "Dog" [] Prefix
```

For a type with non-empty containers we get something a little more interesting. Consider the list type:

```haskell
instance Data a => Data [a] where
  gfoldl _ z []     = z []
  gfoldl k z (x:xs) = z (:) `k` x `k` xs

  toConstr []    = nilConstr
  toConstr (_:_) = consConstr

  gunfold k z c
    = case constrIndex c of
        1 -> z []
        2 -> k (k (z (:)))

  dataTypeOf _ = listDataType

nilConstr :: Constr
nilConstr = mkConstr listDataType "[]" [] Prefix

consConstr :: Constr
consConstr = mkConstr listDataType "(:)" [] Infix

listDataType :: DataType
listDataType = mkDataType "Prelude.[]" [nilConstr,consConstr]
```

Looking at `gfoldl` we see the Data has an implementation of a function for us to walk an applicative over the elements of the constructor by applying a function `k` over each element and applying `z` at the spine. For example look at the instance for a 2-tuple as well:

```
instance (Data a, Data b) => Data (a,b) where
  gfoldl k z (a,b) = z (,) `k` a `k` b

  toConstr (_,_) = tuple2Constr

  gunfold k z c
    = case constrIndex c of
        1 -> k (k (z (,)))

  dataTypeOf _  = tuple2DataType

tuple2Constr :: Constr
tuple2Constr = mkConstr tuple2DataType "(,)" [] Infix

tuple2DataType :: DataType
tuple2DataType = mkDataType "Prelude.(,)" [tuple2Constr]
```

This is pretty neat, now within the same typeclass we have a generic way to introspect any `Data` instance and write logic that depends on the structure and types of its subterms. We can now write a function which allow us to traverse an arbitrary instance Data and twiddle values based on pattern matching on the runtime types. So let's write down a function `over` which increments a `Value` type for both for n-tuples and lists.

```
{-# LANGUAGE DeriveDataTypeable #-}

import Data.Data
import Control.Monad.Identity
import Control.Applicative

data Animal = Cat | Dog deriving (Data, Typeable)

newtype Val = Val Int deriving (Show, Data, Typeable)

incr :: Typeable a => a -> a
incr = maybe id id (cast f)
  where f (Val x) = Val (x * 100)

over :: Data a => a -> a
over x = runIdentity $ gfoldl cont base (incr x)
  where
    cont k d = k <*> (pure $ over d)
    base = pure

example1 :: Constr
example1 = toConstr Dog
```

```
-- Dog

example2 :: DataType
example2 = dataTypeOf Cat
-- DataType {tycon = "Main.Animal", datarep = AlgRep [Cat,Dog]}

example3 :: [Val]
example3 = over [Val 1, Val 2, Val 3]
-- [Val 100,Val 200,Val 300]

example4 :: (Val, Val, Val)
example4 = over (Val 1, Val 2, Val 3)
-- (Val 100,Val 200,Val 300)
```

We can also write generic operations to for instance count the number of parameters in a data type.

```
numHoles :: Data a => a -> Int
numHoles = gmapQl (+) 0 (const 1)

example1 :: Int
example1 = numHoles (1,2,3,4,5,6,7)
-- 7

example2 :: Int
example2 = numHoles (Just 3)
-- 1
```

This method adapts itself well to generic traversals but the types quickly become rather hairy when dealing anymore more complicated involving folds and unsafe coercions.

Generic

The most modern method of doing generic programming uses type families to achieve a better of deriving the structural properties of arbitrary type classes. Generic implements a typeclass with an associated type Rep (Representation) together with a pair of functions that form a 2-sided inverse (isomorphism) for converting to and from the associated type and the derived type in question.

```
class Generic a where
  type Rep a
  from :: a -> Rep a
  to :: Rep a -> a

class Datatype d where
  datatypeName :: t d f a -> String
  moduleName :: t d f a -> String

class Constructor c where
  conName :: t c f a -> String
```

GHC.Generics defines a set of named types for modeling the various structural properties of types in available in Haskell.

```haskell
-- | Sums: encode choice between constructors
infixr 5 :+:
data (:+:) f g p = L1 (f p) | R1 (g p)

-- | Products: encode multiple arguments to constructors
infixr 6 :*:
data (:*:) f g p = f p :*: g p

-- | Tag for M1: datatype
data D
-- | Tag for M1: constructor
data C

-- | Constants, additional parameters and recursion of kind *
newtype K1 i c p = K1 { unK1 :: c }

-- | Meta-information (constructor names, etc.)
newtype M1 i c f p = M1 { unM1 :: f p }

-- | Type synonym for encoding meta-information for datatypes
type D1 = M1 D

-- | Type synonym for encoding meta-information for constructors
type C1 = M1 C
```

Using the deriving mechanics GHC can generate this Generic instance for us mechanically, if we were to write it by hand for a simple type it might look like this:

```haskell
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE TypeFamilies #-}

import GHC.Generics

data Animal
  = Dog
  | Cat

instance Generic Animal where
  type Rep Animal = D1 T_Animal ((C1 C_Dog U1) :+: (C1 C_Cat U1))

  from Dog = M1 (L1 (M1 U1))
  from Cat = M1 (R1 (M1 U1))

  to (M1 (L1 (M1 U1))) = Dog
  to (M1 (R1 (M1 U1))) = Cat

data T_Animal
data C_Dog
```

```
data C_Cat

instance Datatype T_Animal where
  datatypeName _ = "Animal"
  moduleName _ = "Main"

instance Constructor C_Dog where
  conName _ = "Dog"

instance Constructor C_Cat where
  conName _ = "Cat"
```

Use `kind!` in GHCi we can look at the type family `Rep` associated with a Generic instance.

```
λ: :kind! Rep Animal
Rep Animal :: * -> *
= M1 D T_Animal (M1 C C_Dog U1 :+: M1 C C_Cat U1)

λ: :kind! Rep ()
Rep () :: * -> *
= M1 D GHC.Generics.D1() (M1 C GHC.Generics.C1_0() U1)

λ: :kind! Rep [()]
Rep [()] :: * -> *
= M1
    D
    GHC.Generics.D1[]
    (M1 C GHC.Generics.C1_0[] U1
     :+: M1
           C
           GHC.Generics.C1_1[]
           (M1 S NoSelector (K1 R ()) :*: M1 S NoSelector (K1 R [()])))
```

Now the clever bit, instead writing our generic function over the datatype we instead write it over the Rep and then reify the result using `from`. Some for an equivalent version of Haskell's default `Eq` that instead uses generic deriving we could write:

```
class GEq' f where
  geq' :: f a -> f a -> Bool

instance GEq' U1 where
  geq' _ _ = True

instance (GEq c) => GEq' (K1 i c) where
  geq' (K1 a) (K1 b) = geq a b

instance (GEq' a) => GEq' (M1 i c a) where
  geq' (M1 a) (M1 b) = geq' a b

-- Equality for sums.
```

```
instance (GEq' a, GEq' b) => GEq' (a :+: b) where
  geq' (L1 a) (L1 b) = geq' a b
  geq' (R1 a) (R1 b) = geq' a b
  geq' _       _     = False

-- Equality for products.
instance (GEq' a, GEq' b) => GEq' (a :*: b) where
  geq' (a1 :*: b1) (a2 :*: b2) = geq' a1 a2 && geq' b1 b2
```

Now to accommodate the two methods of writing classes (generic-deriving or custom implementations) we can use `DefaultSignatures` extension to allow the user to leave typeclass functions blank and defer to the Generic or to define their own.

```
{-# LANGUAGE DefaultSignatures #-}

class GEq a where
  geq :: a -> a -> Bool

  default geq :: (Generic a, GEq' (Rep a)) => a -> a -> Bool
  geq x y = geq' (from x) (from y)
```

Now anyone using our library need only derive Generic and create an empty instance of our typeclass instance without writing any boilerplate for `GEq`.

See:

- [Andres Loh: Datatype-generic Programming in Haskell](#)
- [generic-deriving](#)

Generic Deriving

Using Generics, we can ask GHC to do lots of non-trivial code generation which works spectacularly well in practice. Some real world examples:

The [hashable](#) library allows us to derive hashing functions.

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics (Generic)
import Data.Hashable

data Color = Red | Green | Blue deriving (Generic, Show)

instance Hashable Color where

example1 :: Int
example1 = hash Red
-- 839657738087498284

example2 :: Int
example2 = hashWithSalt 0xDEADBEEF Red
```

```
-- 62679985574121021
```

The <u>cereal</u> library allows us to automatically derive a binary representation.

```
{-# LANGUAGE DeriveGeneric #-}

import Data.Word
import Data.ByteString
import Data.Serialize

import GHC.Generics

data Val = A [Val] | B [(Val, Val)] | C
  deriving (Generic, Show)

instance Serialize Val where

encoded :: ByteString
encoded = encode (A [B [(C, C)]])
-- "\NUL\NUL\NUL\NUL\NUL\NUL\NUL\NUL\SOH\SOH\NUL\NUL\NUL\NUL\NUL\NUL\NUL\SOH\STX\STX"

bytes :: [Word8]
bytes = unpack encoded
-- [0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,2,2]

decoded :: Either String Val
decoded = decode encoded
```

The <u>aeson</u> library allows us to derive JSON representations for JSON instances.

```
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE OverloadedStrings #-}

import Data.Aeson
import GHC.Generics

data Point = Point { _x :: Double, _y :: Double }
    deriving (Show, Generic)

instance FromJSON Point
instance ToJSON Point

example1 :: Maybe Point
example1 = decode "{\"x\":3.0,\"y\":-1.0}"

example2 = encode $ Point 123.4 20
```

See: <u>A Generic Deriving Mechanism for Haskell</u>

·· · · ·

Uniplate

Uniplate is a generics library for writing traversals and transformation for arbitrary data structures. It is extremely useful for writing AST transformations and rewriting systems.

```
plate   :: from -> Type from to
(|*)    :: Type (to -> from) to -> to -> Type from to
(|-)    :: Type (item -> from) to -> item -> Type from to

descend   :: Uniplate on => (on -> on) -> on -> on
transform :: Uniplate on => (on -> on) -> on -> on
rewrite   :: Uniplate on => (on -> Maybe on) -> on -> on
```

The `descend` function will apply a function to each immediate descendant of an expression and then combines them up into the parent expression.

The `transform` function will perform a single pass bottom-up transformation of all terms in the expression.

The `rewrite` function will perform an exhaustive transformation of all terms in the expression to fixed point, using Maybe to signify termination.

```haskell
import Data.Generics.Uniplate.Direct

data Expr a
  = Fls
  | Tru
  | Var a
  | Not (Expr a)
  | And (Expr a) (Expr a)
  | Or  (Expr a) (Expr a)
  deriving (Show, Eq)

instance Uniplate (Expr a) where
  uniplate (Not f)     = plate Not |* f
  uniplate (And f1 f2) = plate And |* f1 |* f2
  uniplate (Or f1 f2)  = plate Or |* f1 |* f2
  uniplate x           = plate x

simplify :: Expr a -> Expr a
simplify = transform simp
 where
   simp (Not (Not f)) = f
   simp (Not Fls) = Tru
   simp (Not Tru) = Fls
   simp x = x

reduce :: Show a => Expr a -> Expr a
reduce = rewrite cnf
  where
    -- double negation
    cnf (Not (Not p)) = Just p
```

```
    -- de Morgan
    cnf (Not (p `Or` q))  = Just $ (Not p) `And` (Not q)
    cnf (Not (p `And` q)) = Just $ (Not p) `Or` (Not q)

    -- distribute conjunctions
    cnf (p `Or` (q `And` r)) = Just $ (p `Or` q) `And` (p `Or` r)
    cnf ((p `And` q) `Or` r) = Just $ (p `Or` q) `And` (p `Or` r)
    cnf _ = Nothing


example1 :: Expr String
example1 = simplify (Not (Not (Not (Not (Var "a")))))
-- Var "a"

example2 :: [String]
example2 = [a | Var a <- universe ex]
  where
    ex = Or (And (Var "a") (Var "b")) (Not (And (Var "c") (Var "d")))
-- ["a","b","c","d"]

example3 :: Expr String
example3 = reduce $ ((a `And` b) `Or` (c `And` d)) `Or` e
  where
    a = Var "a"
    b = Var "b"
    c = Var "c"
    d = Var "d"
    e = Var "e"
```

Alternatively Uniplate instances can be derived automatically from instances of Data without the need to explicitly write a Uniplate instance. This approach carries a slight amount of overhead over an explicit hand-written instance.

```
import Data.Data
import Data.Typeable
import Data.Generics.Uniplate.Data

data Expr a
  = Fls
  | Tru
  | Lit a
  | Not (Expr a)
  | And (Expr a) (Expr a)
  | Or (Expr a) (Expr a)
  deriving (Data, Typeable, Show, Eq)
```

Biplate

Biplates generalize plates where the target type isn't necessarily the same as the source, it uses multiparameter typeclasses to indicate the type sub of the sub-target. The Uniplate functions all have an equivalent generalized biplate form.

```haskell
descendBi   :: Biplate from to => (to -> to) -> from -> from
transformBi :: Biplate from to => (to -> to) -> from -> from
rewriteBi   :: Biplate from to => (to -> Maybe to) -> from -> from

descendBiM   :: (Monad m, Biplate from to) => (to -> m to) -> from -> m from
transformBiM :: (Monad m, Biplate from to) => (to -> m to) -> from -> m from
rewriteBiM   :: (Monad m, Biplate from to) => (to -> m (Maybe to)) -> from -> m from
```

```haskell
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleContexts #-}

import Data.Generics.Uniplate.Direct

type Name = String

data Expr
  = Var Name
  | Lam Name Expr
  | App Expr Expr
  deriving Show

data Stmt
  = Decl [Stmt]
  | Let Name Expr
  deriving Show

instance Uniplate Expr where
  uniplate (Var x  ) = plate Var |- x
  uniplate (App x y) = plate App |* x |* y
  uniplate (Lam x y) = plate Lam |- x |* y

instance Biplate Expr Expr where
  biplate = plateSelf

instance Uniplate Stmt where
  uniplate (Decl x  ) = plate Decl ||* x
  uniplate (Let x y) = plate Let |-  x |- y

instance Biplate Stmt Stmt where
  biplate = plateSelf

instance Biplate Stmt Expr where
  biplate (Decl x) = plate Decl ||+ x
  biplate (Let x y) = plate Let |- x |* y

rename :: Name -> Name -> Expr -> Expr
rename from to = rewrite f
  where
    f (Var a) | a == from = Just (Var to)
    f (Lam a b) | a == from = Just (Lam to b)
```

```
    f _ = Nothing

s, k, sk :: Expr
s = Lam "x" (Lam "y" (Lam "z" (App (App (Var "x") (Var "z")) (App (Var "y") (Var "z"))
k = Lam "x" (Lam "y" (Var "x"))
sk = App s k


m :: Stmt
m = descendBi f $ Decl [ (Let "s" s) , Let "k" k , Let "sk" sk ]
  where
    f = rename "x" "a"
      . rename "y" "b"
      . rename "z" "c"
```

Mathematics

Numeric Tower

Haskell's numeric tower is unusual and the source of some confusion for novices. Haskell is one of the few languages to incorporate statically typed overloaded literals without a mechanism for "coercions" often found in other languages.

To add to the confusion numerical literals in Haskell are desugared into a function from a numeric typeclass which yields a polymorphic value that can be instantiated to any instance of the (Num) or (Fractional) typeclass at the call-site, depending on the inferred type.

To use a blunt metaphor, we're effectively placing an object in a hole and the size and shape of the hole defines the object you place there. This is very different than in other languages where a numeric literal like (2.718) is hard coded in the compiler to be a specific type (double or something) and you cast the value at runtime to be something smaller or larger as needed.

```
42 :: Num a => a
fromInteger (42 :: Integer)

2.71 :: Fractional a => a
fromRational (2.71 :: Rational)
```
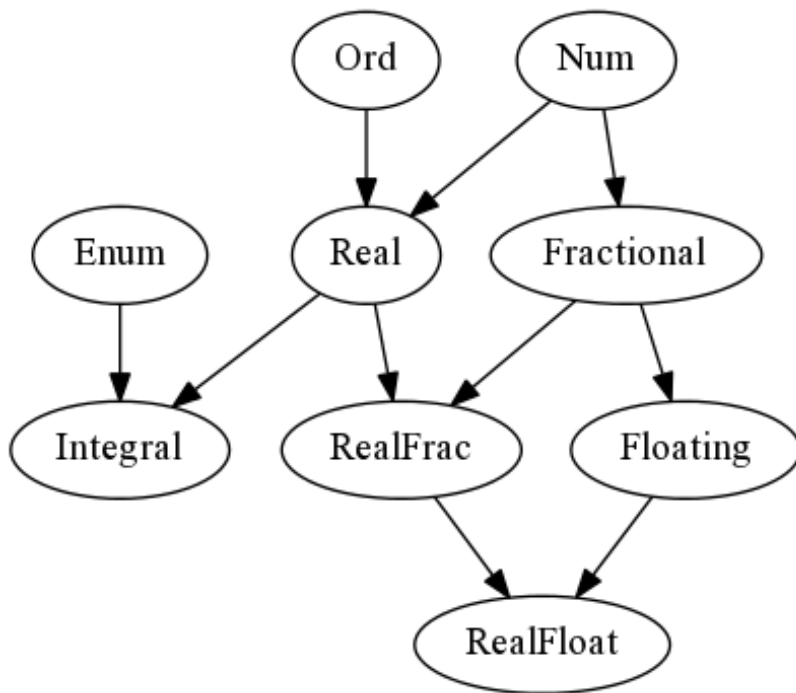
The numeric typeclass hierarchy is defined as such:

```
class Num a
class (Num a, Ord a) => Real a
class Num a => Fractional a
class (Real a, Enum a) => Integral a
class (Real a, Fractional a) => RealFrac a
class Fractional a => Floating a
class (RealFrac a, Floating a) => RealFloat a
```

Conversions between concrete numeric types (from : left column, to : top row) is accomplished with several generic functions.

	Double	Float	Int	Word	Integer	Rational
Double	id	fromRational	truncate	truncate	truncate	toRational
Float	fromRational	id	truncate	truncate	truncate	toRational
Int	fromIntegral	fromIntegral	id	fromIntegral	fromIntegral	fromIntegral
Word	fromIntegral	fromIntegral	fromIntegral	id	fromIntegral	fromIntegral
Integer	fromIntegral	fromIntegral	fromIntegral	fromIntegral	id	fromIntegral
Rational	fromRatoinal	fromRational	truncate	truncate	truncate	id

Integer

The `Integer` type in GHC is implemented by the GMP (`libgmp`) arbitrary precision arithmetic library. Unlike the `Int` type the size of Integer values is bounded only by the available memory. Most notably `libgmp` is one of the few libraries that compiled Haskell binaries are dynamically linked against.

An alternative library `integer-simple` can be linked in place of libgmp.

See: GHC, primops and exorcising GMP

Complex

Haskell supports arithmetic with complex numbers via a Complex datatype. The first argument is the real part, while the second is the imaginary.

```
-- 1 + 2i
let complex = 1 :+ 2
```

```
data Complex a = a :+ a
mkPolar :: RealFloat a => a -> a -> Complex a
```

```
λ: 0 :+ 1
0 :+ 1 :: Complex Integer

λ: (0 :+ 1) + (1 :+ 0)
1.0 :+ 1.0 :: Complex Integer

λ: exp (0 :+ 2 * pi)
1.0 :+ (-2.4492935982947064e-16) :: Complex Double

λ: mkPolar 1 (2*pi)
1.0 :+ (-2.4492935982947064e-16) :: Complex Double

λ: let f x n = (cos x :+ sin x)^n
λ: let g x n = cos (n*x) :+ sin (n*x)
```

Scientific

```
scientific :: Integer -> Int -> Scientific
fromFloatDigits :: RealFloat a => a -> Scientific
```

Scientific provides arbitrary-precision numbers represented using scientific notation. The constructor takes an arbitrarily sized Integer argument for the digits and an Int for the exponent. Alternatively the value can be parsed from a String or coerced from either Double/Float.

```
import Data.Scientific

c, h, g, a, k :: Scientific
c = scientific 299792458 (0)    -- Speed of light
h = scientific 662606957 (-42) -- Planck's constant
g = scientific 667384    (-16) -- Gravitational constant
a = scientific 729735257 (-11) -- Fine structure constant
k = scientific 268545200 (-9)  -- Khinchin Constant

tau :: Scientific
tau = fromFloatDigits (2*pi)

maxDouble64 :: Double
maxDouble64 = read "1.7976931348623159e308"
-- Infinity

maxScientific :: Scientific
maxScientific = read "1.7976931348623159e308"
-- 1.7976931348623159e308
```

Statistics

```haskell
import Data.Vector
import Statistics.Sample

import Statistics.Distribution.Normal
import Statistics.Distribution.Poisson
import qualified Statistics.Distribution as S

s1 :: Vector Double
s1 = fromList [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

s2 :: PoissonDistribution
s2 = poisson 2.5

s3 :: NormalDistribution
s3 = normalDistr mean stdDev
  where
    mean   = 1
    stdDev = 1

descriptive = do
  print $ range s1
  -- 9.0
  print $ mean s1
  -- 5.5
  print $ stdDev s1
  -- 3.0276503540974917
  print $ variance s1
  -- 8.25
  print $ harmonicMean s1
  -- 3.414171521474055
  print $ geometricMean s1
  -- 4.5287286881167645

discrete = do
  print $ S.cumulative s2 0
  -- 8.208499862389884e-2
  print $ S.mean s2
  -- 2.5
  print $ S.variance s2
  -- 2.5
  print $ S.stdDev s2
  -- 1.5811388300841898

continuous = do
  print $ S.cumulative s3 0
  -- 0.15865525393145707
  print $ S.quantile s3 0.5
  -- 1.0
  print $ S.density s3 0
  -- 0.24197072451914334
```

```
    print $ S.mean s3
    -- 1.0
    print $ S.variance s3
    -- 1.0
    print $ S.stdDev s3
    -- 1.0
```

Constructive Reals

Instead of modeling the real numbers on finite precision floating point numbers we alternatively work with ⟨Num⟩ which internally manipulate the power series expansions for the expressions when performing operations like arithmetic or transcendental functions without losing precision when performing intermediate computations. Then we simply slice off a fixed number of terms and approximate the resulting number to a desired precision. This approach is not without its limitations and caveats (notably that it may diverge) but works quite well in practice.

```
exp(x)    = 1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 ...
sqrt(1+x) = 1 + 1/2*x - 1/8*x^2 + 1/16*x^3 - 5/128*x^4 + 7/256*x^5 ...
atan(x)   = x - 1/3*x^3 + 1/5*x^5 - 1/7*x^7 + 1/9*x^9 - 1/11*x^11 ...
pi        = 16 * atan (1/5) - 4 * atan (1/239)
```

```haskell
import Data.Number.CReal

-- algebraic
phi :: CReal
phi = (1 + sqrt 5) / 2

-- transcendental
ramanujan :: CReal
ramanujan = exp (pi * sqrt 163)

main :: IO ()
main = do
  putStrLn $ showCReal 30 pi
  -- 3.141592653589793238462643383279
  putStrLn $ showCReal 30 phi
  -- 1.618033988749894848204586834366
  putStrLn $ showCReal 15 ramanujan
  -- 262537412640768743.99999999999925
```

SAT Solvers

A collection of constraint problems known as satisfiability problems show up in a number of different disciplines from type checking to package management. Simply put a satisfiability problem attempts to find solutions to a statement of conjoined conjunctions and disjunctions in terms of a series of variables. For example:

```
(A ∨ ¬B ∨ C) ∧ (B ∨ D ∨ E) ∧ (D ∨ F)
```

To use the picosat library to solve this, it can be written as zero-terminated lists of integers and fed to the solver according to a number-to-variable relation:

```
1 -2 3    -- (A ∨ ¬B ∨ C)
2 4 5     -- (B ∨ D ∨ E)
4 6       -- (D ∨ F)
```

```haskell
import Picosat

main :: IO [Int]
main = do
  solve [[1, -2, 3], [2,4,5], [4,6]]
  -- Solution [1,-2,3,4,5,6]
```

The SAT solver itself can be used to solve satisfiability problems with millions of variables in this form and is finely tuned.

See:

- picosat

SMT Solvers

A generalization of the SAT problem to include predicates other theories gives rise to the very sophisticated domain of "Satisfiability Modulo Theory" problems. The existing SMT solvers are very sophisticated projects (usually bankrolled by large institutions) and usually have to called out to via foreign function interface or via a common interface called SMT-lib. The two most common of use in Haskell are `cvc4` from Stanford and `z3` from Microsoft Research.

The SBV library can abstract over different SMT solvers to allow us to express the problem in an embedded domain language in Haskell and then offload the solving work to the third party library.

TODO: Talk about SBV

See:

- cvc4
- z3

Data Structures

Map

```haskell
import qualified Data.Map as Map

kv :: Map.Map Integer String
kv = Map.fromList [(1, "a"), (2, "b")]

lkup :: Integer -> String -> String
lkup key def =
  case Map.lookup key kv of
    Just val -> val
```

```
        Nothing  -> def
```

Tree

```haskell
import Data.Tree

{-

   A
  / \
 B   C
    / \
   D   E

-}

tree :: Tree String
tree = Node "A" [Node "B" [], Node "C" [Node "D" [], Node "E" []]]

postorder :: Tree a -> [a]
postorder (Node a ts) = elts ++ [a]
  where elts = concat (map postorder ts)

preorder :: Tree a -> [a]
preorder (Node a ts) = a : elts
  where elts = concat (map preorder ts)

ex1 = drawTree tree
ex2 = drawForest (subForest tree)
ex3 = flatten tree
ex4 = levels tree
ex5 = preorder tree
ex6 = postorder tree
```

Set

```haskell
import qualified Data.Set as Set

set :: Set.Set Integer
set = Set.fromList [1..1000]

memtest :: Integer -> Bool
memtest elt = Set.member elt set
```

Vector

Vectors are high performance single dimensional arrays that come come in six variants, two for each of the following types of a mutable and an immutable variant.

- Data.Vector
- Data.Vector.Storable
- Data.Vector.Unboxed

The most notable feature of vectors is constant time memory access with (`(!)`) as well as variety of efficient map, fold and scan operations on top of a fusion framework that generates surprisingly optimal code.

```haskell
fromList :: [a] -> Vector a
toList :: Vector a -> [a]
(!) :: Vector a -> Int -> a
map :: (a -> b) -> Vector a -> Vector b
foldl :: (a -> b -> a) -> a -> Vector b -> a
scanl :: (a -> b -> a) -> a -> Vector b -> Vector a
zipWith :: (a -> b -> c) -> Vector a -> Vector b -> Vector c
iterateN :: Int -> (a -> a) -> a -> Vector a
```

```haskell
import Data.Vector.Unboxed as V

norm ::  Vector Double -> Double
norm = sqrt . V.sum . V.map (\x -> x*x)

example1 :: Double
example1 = norm $ V.iterateN 100000000 (+1) 0.0
```

See: <u>Numerical Haskell: A Vector Tutorial</u>

Mutable Vectors

```haskell
freeze :: MVector (PrimState m) a -> m (Vector a)
thaw :: Vector a -> MVector (PrimState m) a
```

Within the IO monad we can perform arbitrary read and writes on the mutable vector with constant time reads and writes. When needed a static Vector can be created to/from the `MVector` using the freeze/thaw functions.

```haskell
import GHC.Prim
import Control.Monad
import Control.Monad.ST
import Control.Monad.Primitive

import Data.Vector.Unboxed (freeze)
import Data.Vector.Unboxed.Mutable
import qualified Data.Vector.Unboxed as V
```

```haskell
example :: PrimMonad m => m (V.Vector Int)
example = do
  v <- new 10
  forM_ [0..9] $ \i ->
      write v i (2*i)
  freeze v

-- vector computation in IO
vecIO :: IO (V.Vector Int)
vecIO = example

-- vector computation in ST
vecST :: ST s (V.Vector Int)
vecST = example


main :: IO ()
main = do
  vecIO >>= print
  print $ runST vecST
```

Unordered-Containers

```haskell
fromList :: (Eq k, Hashable k) => [(k, v)] -> HashMap k v
lookup :: (Eq k, Hashable k) => k -> HashMap k v -> Maybe v
insert :: (Eq k, Hashable k) => k -> v -> HashMap k v -> HashMap k v
```

Both the `HashMap` and `HashSet` are purely functional data structures that are drop in replacements for the `containers` equivalents but with more efficient space and time performance. Additionally all stored elements must have a `Hashable` instance.

```haskell
import qualified Data.HashSet as S
import qualified Data.HashMap.Lazy as M

example1 :: M.HashMap Int Char
example1 = M.fromList $ zip [1..10] ['a'..]

example2 :: S.HashSet Int
example2 = S.fromList [1..10]
```

See: Johan Tibell: Announcing Unordered Containers

Hashtables

Hashtables provides hashtables with efficient lookup within the ST or IO monad.

```haskell
import Prelude hiding (lookup)
```

```haskell
import Control.Monad.ST
import Data.HashTable.ST.Basic

-- Hashtable parameterized by ST "thread"
type HT s = HashTable s String String

set :: ST s (HT s)
set = do
  ht <- new
  insert ht "key" "value1"
  return ht

get :: HT s -> ST s (Maybe String)
get ht = do
  val <- lookup ht "key"
  return val

example :: Maybe String
example = runST (set >>= get)
```

```haskell
new :: ST s (HashTable s k v)
insert :: (Eq k, Hashable k) => HashTable s k v -> k -> v -> ST s ()
lookup :: (Eq k, Hashable k) => HashTable s k v -> k -> ST s (Maybe v)
```

Graphs

The Graph module in the containers library is a somewhat antiquated API for working with directed graphs. A little bit of data wrapping makes it a little more straightforward to use. The library is not necessarily well-suited for large graph-theoretic operations but is perfectly fine for example, to use in a typechecker which need to resolve strongly connected components of the module definition graph.

```haskell
import Data.Tree
import Data.Graph

data Grph node key = Grph
  { _graph :: Graph
  , _vertices :: Vertex -> (node, key, [key])
  }

fromList :: Ord key => [(node, key, [key])] -> Grph node key
fromList = uncurry Grph . graphFromEdges'

vertexLabels :: Functor f => Grph b t -> (f Vertex) -> f b
vertexLabels g = fmap (vertexLabel g)

vertexLabel :: Grph b t -> Vertex -> b
vertexLabel g = (\(vi, _, _) -> vi) . (_vertices g)

-- Topologically sort graph
```

```
topo' :: Grph node key -> [node]
topo' g = vertexLabels g $ topSort (_graph g)

-- Strongly connected components of graph
scc' :: Grph node key -> [[node]]
scc' g = fmap (vertexLabels g . flatten) $ scc (_graph g)
```

So for example we can construct a simple graph:
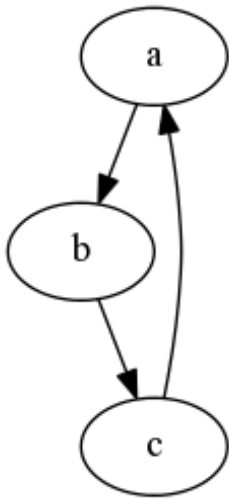

```
ex1 :: [(String, String, [String])]
ex1 = [
    ("a","a",["b"]),
    ("b","b",["c"]),
    ("c","c",["a"])
  ]

ts1 :: [String]
ts1 = topo' (fromList ex1)
-- ["a","b","c"]

sc1 :: [[String]]
sc1 = scc' (fromList ex1)
-- [["a","b","c"]]
```

Or with two strongly connected subgraphs:

```haskell
ex2 :: [(String, String, [String])]
ex2 = [
    ("a","a",["b"]),
    ("b","b",["c"]),
    ("c","c",["a"]),

    ("d","d",["e"]),
    ("e","e",["f", "e"]),
    ("f","f",["d", "e"])
  ]


ts2 :: [String]
ts2 = topo' (fromList ex2)
-- ["d","e","f","a","b","c"]

sc2 :: [[String]]
sc2 = scc' (fromList ex2)
-- [["d","e","f"],["a","b","c"]]
```
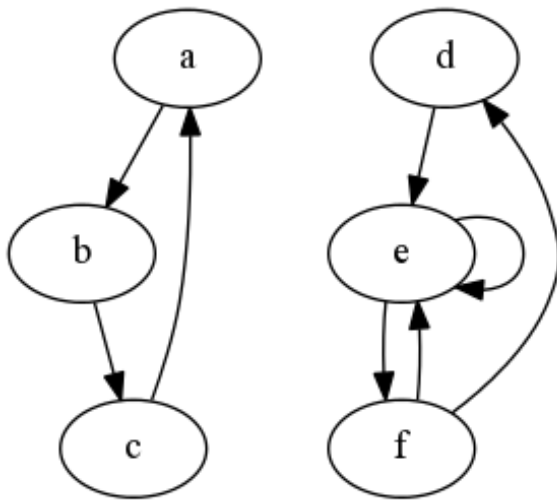
See: GraphSCC

Graph Theory

The (fgl) library provides are more efficient graph structure and a wide variety of common graph-theoretic operations. For example calculating the dominance frontier of a graph shows up quite frequently in control flow analysis for compiler design.

```haskell
import qualified Data.Graph.Inductive as G

cyc3 :: G.Gr Char String
cyc3 = G.buildGr
        [([("ca",3)],1,'a',[("ab",2)]),
                    ([],2,'b',[("bc",3)]),
                    ([],3,'c',[])]

-- Loop query
```

```haskell
ex1 :: Bool
ex1 = G.hasLoop x

-- Dominators
ex2 :: [(G.Node, [G.Node])]
ex2 = G.dom x 0
```

```haskell
x :: G.Gr Int ()
x = G.insEdges edges gr
  where
  gr = G.insNodes nodes G.empty
  edges = [(0,1,()), (0,2,()), (2,1,()), (2,3,())]
  nodes = zip [0,1 ..] [2,3,4,1]
```
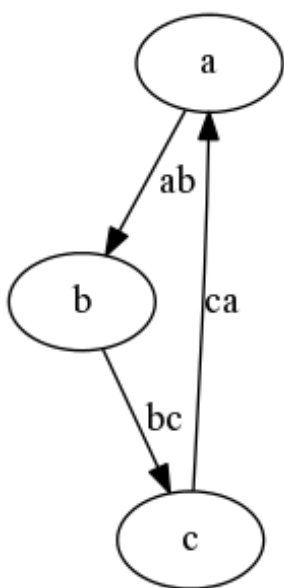

DList

A dlist is a list-like structure that is optimized for O(1) append operations, internally it uses a Church encoding of the list structure. It is specifically suited for operations which are append-only and need only access it when manifesting the entire structure. It is particularly well-suited for use in the Writer monad.

```haskell
import Data.DList
import Control.Monad
import Control.Monad.Writer

logger :: Writer (DList Int) ()
logger = replicateM_ 100000 $ tell (singleton 0)
```

Sequence

The sequence data structure behaves structurally similar to list but is optimized for append/prepend operations and traversal.

```haskell
import Data.Sequence

a :: Seq Int
a = fromList [1,2,3]

a0 :: Seq Int
a0 = a |> 4
-- [1,2,3,4]

a1 :: Seq Int
a1 = 0 <| a
-- [0,1,2,3]
```

Matrices and HBlas

Just as in C when working with n-dimensional matrices we'll typically overlay the high-level matrix structure onto an unboxed contiguous block of memory with index functions which perform the coordinate translations to calculate offsets. The two most common layouts are:

- Row-major order
- Column-major order

Which are best illustrated.

The calculations have a particularly nice implementation in Haskell in terms of scans over indices.

```haskell
import qualified Data.Vector as V

data Order = RowMajor | ColMajor

rowMajor :: [Int] -> [Int]
rowMajor = scanr (*) 1 . tail

colMajor :: [Int] -> [Int]
colMajor = init . scanl (*) 1

data Matrix a = Matrix
  { _dims  :: [Int]
  , _elts  :: V.Vector a
  , _order :: Order
  }
```

```haskell
fromList :: [Int] -> Order -> [a] -> Matrix a
fromList sh order elts =
  if product sh == length elts
  then Matrix sh (V.fromList elts) order
  else error "dimensions don't match"

indexTo :: [Int] -> Matrix a -> a
indexTo ix mat = boundsCheck offset
  where
    boundsCheck n =
      if 0 <= n && n < V.length (_elts mat)
      then V.unsafeIndex (_elts mat) offset
      else error "out of bounds"
    ordering = case _order mat of
      RowMajor -> rowMajor
      ColMajor -> colMajor
    offset = sum $ zipWith (*) ix (ordering (_dims mat))

matrix :: Order -> Matrix Int
matrix order = fromList [4,4] order [1..16]

ex1 :: [Int]
ex1 = rowMajor [1,2,3,4]
-- [24,12,4,1]

ex2 :: [Int]
ex2 = colMajor [1,2,3,4]
-- [1,1,2,6]

ex3 :: Int
ex3 = indexTo [1,3] (matrix RowMajor)
-- 8

ex4 :: Int
ex4 = indexTo [1,3] (matrix ColMajor)
-- 14
```

Unboxed matrices of this type can also be passed to C or Fortran libraries such BLAS or LAPACK linear algebra libraries. The `hblas` package wraps many of these routines and forms the low-level wrappers for higher level-libraries that need access to these foreign routines.

For example the <u>dgemm</u> routine takes two pointers to a sequence of `double` values of two matrices of size `(m × k)` and `(k × n)` and performs efficient matrix multiplication writing the resulting data through a pointer to a `(m × n)` matrix.

```haskell
import Foreign.Storable
import Numerical.HBLAS.BLAS
import Numerical.HBLAS.MatrixTypes

-- Generate the constant mutable square matrix of the given type and dimensions.
constMatrix :: Storable a => Int -> a -> IO (IODenseMatrix Row a)
constMatrix n k = generateMutableDenseMatrix SRow (n,n) (const k)
```

```haskell
example_dgemm :: IO ()
example_dgemm = do
    left  <- constMatrix 2 (2 :: Double)
    right <- constMatrix 2 (3 :: Double)
    out   <- constMatrix 2 (0 :: Double)

    dgemm NoTranspose NoTranspose 1.0 1.0 left right out

    resulting <- mutableVectorToList $ _bufferDenMutMat out
    print resulting
```

See: hblas

FFI

Pure Functions

Wrapping pure C functions with primitive types is trivial.

```c
/* $(CC) -c simple.c -o simple.o */

int example(int a, int b)
{
  return a + b;
}
```

```haskell
-- ghc simple.o simple_ffi.hs -o simple_ffi
{-# LANGUAGE ForeignFunctionInterface #-}

import Foreign.C.Types

foreign import ccall safe "example" example
    :: CInt -> CInt -> CInt

main = print (example 42 27)
```

Storable Arrays

There exists a `Storable` typeclass that can be used to provide low-level access to the memory underlying Haskell values. `Ptr` objects in Haskell behave much like C pointers although arithmetic with them is in terms of bytes only, not the size of the type associated with the pointer (this differs from C).

The Prelude defines Storable interfaces for most of the basic types as well as types in the `Foreign.C` library.

```haskell
class Storable a where
```

```haskell
sizeOf :: a -> Int
alignment :: a -> Int
peek :: Ptr a -> IO a
poke :: Ptr a -> a -> IO ()
```

To pass arrays from Haskell to C we can again use Storable Vector and several unsafe operations to grab a foreign pointer to the underlying data that can be handed off to C. Once we're in C land, nothing will protect us from doing evil things to memory!

```c
/* $(CC) -c qsort.c -o qsort.o */
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

void sort(int *xs, int beg, int end)
{
    if (end > beg + 1) {
        int piv = xs[beg], l = beg + 1, r = end;

        while (l < r) {
            if (xs[l] <= piv) {
                l++;
            } else {
                swap(&xs[l], &xs[--r]);
            }
        }

        swap(&xs[--l], &xs[beg]);
        sort(xs, beg, l);
        sort(xs, r, end);
    }
}
```

```haskell
-- ghc qsort.o ffi.hs -o ffi
{-# LANGUAGE ForeignFunctionInterface #-}

import Foreign.Ptr
import Foreign.C.Types

import qualified Data.Vector.Storable as V
import qualified Data.Vector.Storable.Mutable as VM

foreign import ccall safe "sort" qsort
    :: Ptr a -> CInt -> CInt -> IO ()

main :: IO ()
main = do
```

```
let vs = V.fromList ([1,3,5,2,1,2,5,9,6] :: [CInt])
v <- V.thaw vs
VM.unsafeWith v $ \ptr -> do
  qsort ptr 0 9
out <- V.freeze v
print out
```

The names of foreign functions from a C specific header file can be qualified.

```
foreign import ccall unsafe "stdlib.h malloc"
    malloc :: CSize -> IO (Ptr a)
```

Prepending the function name with a & allows us to create a reference to the function pointer itself.

```
foreign import ccall unsafe "stdlib.h &malloc"
    malloc :: FunPtr a
```

Function Pointers

Using the above FFI functionality, it's trivial to pass C function pointers into Haskell, but what about the inverse passing a function pointer to a Haskell function into C using `foreign import ccall "wrapper"` .

```
#include <stdio.h>

void invoke(void *fn(int))
{
  int n = 42;
  printf("Inside of C, now we'll call Haskell.\n");
  fn(n);
  printf("Back inside of C again.\n");
}
```

```
{-# LANGUAGE ForeignFunctionInterface #-}

import Foreign
import System.IO
import Foreign.C.Types(CInt(..))

foreign import ccall "wrapper"
  makeFunPtr :: (CInt -> IO ()) -> IO (FunPtr (CInt -> IO ()))

foreign import ccall "pointer.c invoke"
  invoke :: FunPtr (CInt -> IO ()) -> IO ()

fn :: CInt -> IO ()
```

```
fn n = do
  putStrLn "Hello from Haskell, here's a number passed between runtimes:"
  print n
  hFlush stdout

main :: IO ()
main = do
  fptr <- makeFunPtr fn
  invoke fptr
```

Will yield the following output:

```
Inside of C, now we'll call Haskell
Hello from Haskell, here's a number passed between runtimes:
42
Back inside of C again.
```

Concurrency

The definitive reference on concurrency and parallelism in Haskell is Simon Marlow's text. This will section will just gloss over these topics because they are far better explained in this book.

See: Parallel and Concurrent Programming in Haskell

```
forkIO :: IO () -> IO ThreadId
```

Haskell threads are extremely cheap to spawn, using only 1.5KB of RAM depending on the platform and are much cheaper than a pthread in C. Calling forkIO 10^6 times completes just short of a 1s. Additionally, functional purity in Haskell also guarantees that a thread can almost always be terminated even in the middle of a computation without concern.

See: The Scheduler

Sparks

The most basic "atom" of parallelism in Haskell is a spark. It is a hint to the GHC runtime that a computation can be evaluated to weak head normal form in parallel.

```
rpar :: a -> Eval a
rseq :: Strategy a
rdeepseq :: NFData a => Strategy a

runEval :: Eval a -> a
```

`rpar a` spins off a separate spark that evolutes a to weak head normal form and places the computation in the spark pool. When the runtime determines that there is an available CPU to evaluate the computation it will evaluate (*convert*) the spark. If the main thread of the program is the evaluator for the spark, the spark is said to have *fizzled*. Fizzling is generally bad and indicates that the logic or

parallelism strategy is not well suited to the work that is being evaluated.

The spark pool is also limited (but user-adjustable) to a default of 8000 (as of GHC 7.8.3). Sparks that are created beyond that limit are said to *overflow*.

```haskell
-- Evaluates the arguments to f in parallel before application.
par2 f x y = x `rpar` y `rpar` f x y
```

An argument to `rseq` forces the evaluation of a spark before evaluation continues.

Action	Description
`Fizzled`	The resulting value has already been evaluated by the main thread so the spark need not be converted.
`Dud`	The expression has already been evaluated, the computed value is returned and the spark is not converted.
`GC'd`	The spark is added to the spark pool but the result is not referenced, so it is garbage collected.
`Overflowed`	Insufficient space in the spark pool when spawning.

The parallel runtime is necessary to use sparks, and the resulting program must be compiled with `-threaded`. Additionally the program itself can be specified to take runtime options with `-rtsopts` such as the number of cores to use.

```bash
ghc -threaded -rtsopts program.hs
./program +RTS -s N8 -- use 8 cores
```

The runtime can be asked to dump information about the spark evaluation by passing the `-s` flag.

```
$ ./spark +RTS -N4 -s

                            Tot time (elapsed)  Avg pause  Max pause
  Gen  0        5 colls,    5 par    0.02s    0.01s      0.0017s    0.0048s
  Gen  1        3 colls,    2 par    0.00s    0.00s      0.0004s    0.0007s

  Parallel GC work balance: 1.83% (serial 0%, perfect 100%)

  TASKS: 6 (1 bound, 5 peak workers (5 total), using -N4)

  SPARKS: 20000 (20000 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
```

The parallel computations themselves are sequenced in the `Eval` monad, whose evaluation with `runEval` is itself a pure computation.

```haskell
example :: (a -> b) -> a -> a -> (b, b)
example f x y = runEval $ do
  a <- rpar $ f x
  b <- rpar $ f y
  rseq a
```

```
  rseq b
  return (a, b)
```

Threadscope

Passing the flag `-l` generates the eventlog which can be rendered with the threadscope library.

```
$ ghc -O2 -threaded -rtsopts -eventlog Example.hs
$ ./program +RTS -N4 -l
$ threadscope Example.eventlog
```
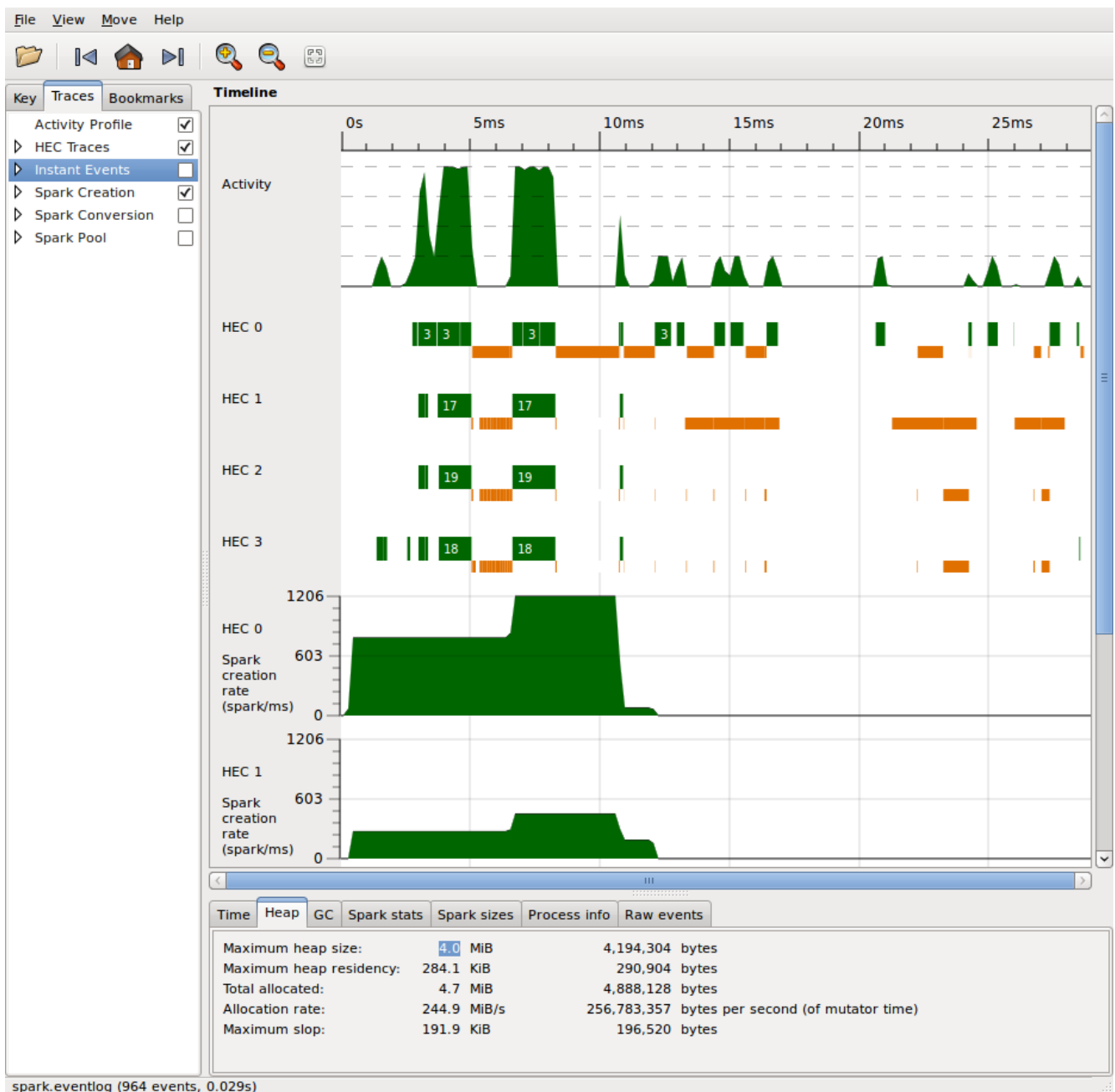

See Simon Marlows's *Parallel and Concurrent Programming in Haskell* for a detailed guide on interpreting and profiling using Threadscope.

See:

- Performance profiling with ghc-events-analyze

Strategies

```haskell
type Strategy a = a -> Eval a
using :: a -> Strategy a -> a
```

Sparks themselves form the foundation for higher level parallelism constructs known as `strategies` which adapt spark creation to fit the computation or data structure being evaluated. For instance if we wanted to evaluate both elements of a tuple in parallel we can create a strategy which uses sparks to evaluate both sides of the tuple.

```haskell
import Control.Parallel.Strategies

parPair' :: Strategy (a, b)
parPair' (a, b) = do
  a' <- rpar a
  b' <- rpar b
  return (a', b')

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

serial :: (Int, Int)
serial   = (fib 30, fib 31)

parallel :: (Int, Int)
parallel = runEval . parPair' $ (fib 30, fib 31)
```

This pattern occurs so frequently the combinator `using` can be used to write it equivalently in operator-like form that may be more visually appealing to some.

```haskell
using :: a -> Strategy a -> a
x `using` s = runEval (s x)

parallel ::: (Int, Int)
parallel = (fib 30, fib 31) `using` parPair
```

For a less contrived example consider a parallel `parmap` which maps a pure function over a list of a values in parallel.

```haskell
import Control.Parallel.Strategies

parMap' :: (a -> b) -> [a] -> Eval [b]
parMap' f [] = return []
parMap' f (a:as) = do
```

```
   b  <- rpar (f a)
   bs <- parMap' f as
   return (b:bs)

result :: [Int]
result = runEval $ parMap' (+1) [1..1000]
```

The functions above are quite useful, but will break down if evaluation of the arguments needs to be parallelized beyond simply weak head normal form. For instance if the arguments to rpar is a nested constructor we'd like to parallelize the entire section of work in evaluated the expression to normal form instead of just the outer layer. As such we'd like to generalize our strategies so the the evaluation strategy for the arguments can be passed as an argument to the strategy.

Control.Parallel.Strategies contains a generalized version of rpar which embeds additional evaluation logic inside the rpar computation in Eval monad.

```
rparWith :: Strategy a -> Strategy a
```

Using the deepseq library we can now construct a Strategy variant of rseq that evaluates to full normal form.

```
rdeepseq :: NFData a => Strategy a
rdeepseq x = rseq (force x)
```

We now can create a "higher order" strategy that takes two strategies and itself yields a a computation which when evaluated uses the passed strategies in its scheduling.

```
import Control.DeepSeq
import Control.Parallel.Strategies

evalPair :: Strategy a -> Strategy b -> Strategy (a, b)
evalPair sa sb (a, b) = do
  a' <- sa a
  b' <- sb b
  return (a', b')

parPair :: Strategy a -> Strategy b -> Strategy (a, b)
parPair sa sb = evalPair (rparWith sa) (rparWith sb)

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

serial :: ([Int], [Int])
serial = (a, b)
  where
    a = fmap fib [0..30]
    b = fmap fib [1..30]
```

```
parallel :: ([Int], [Int])
parallel = (a, b) `using` evalPair rdeepseq rdeepseq
  where
    a = fmap fib [0..30]
    b = fmap fib [1..30]
```

These patterns are implemented in the Strategies library along with several other general forms and combinators for combining strategies to fit many different parallel computations.

```
parTraverse :: Traversable t => Strategy a -> Strategy (t a)
dot :: Strategy a -> Strategy a -> Strategy a
($||) :: (a -> b) -> Strategy a -> a -> b
(.||) :: (b -> c) -> Strategy b -> (a -> b) -> a -> c
```

See:

- Control.Concurent.Strategies

STM

```
atomically :: STM a -> IO a
orElse :: STM a -> STM a -> STM a
retry :: STM a

newTVar :: a -> STM (TVar a)
newTVarIO :: a -> IO (TVar a)
writeTVar :: TVar a -> a -> STM ()
readTVar :: TVar a -> STM a

modifyTVar :: TVar a -> (a -> a) -> STM ()
modifyTVar' :: TVar a -> (a -> a) -> STM ()
```

Software Transactional Memory is a technique for guaranteeing atomicity of values in parallel computations, such that all contexts view the same data when read and writes are guaranteed never to result in inconsistent states.

The strength of Haskell's purity guarantees that transactions within STM are pure and can always be rolled back if a commit fails.

```
import Control.Monad
import Control.Concurrent
import Control.Concurrent.STM

type Account = TVar Double

transfer :: Account -> Account -> Double -> STM ()
transfer from to amount = do
```

```
    available <- readTVar from
    when (amount > available) retry

    modifyTVar from (+ (-amount))
    modifyTVar to   (+ amount)

-- Threads are scheduled non-deterministically.
actions :: Account -> Account -> [IO ThreadId]
actions a b = map forkIO [
    -- transfer to
      atomically (transfer a b 10)
    , atomically (transfer a b (-20))
    , atomically (transfer a b 30)

    -- transfer back
    , atomically (transfer a b (-30))
    , atomically (transfer a b 20)
    , atomically (transfer a b (-10))
  ]

main :: IO ()
main = do
  accountA <- atomically $ newTVar 60
  accountB <- atomically $ newTVar 0

  sequence_ (actions accountA accountB)

  balanceA <- atomically $ readTVar accountA
  balanceB <- atomically $ readTVar accountB

  print $ balanceA == 60
  print $ balanceB == 0
```

See: Beautiful Concurrency

Monad Par

Using the Par monad we express our computation as a data flow graph which is scheduled in order of the connections between forked computations which exchange resulting computations with `IVar` .

```
new :: Par (IVar a)
put :: NFData a => IVar a -> a -> Par ()
get :: IVar a -> Par a
fork :: Par () -> Par ()
spawn :: NFData a => Par a -> Par (IVar a)
```

f x   g x

a + b

f (a + b)   g (a + b)

(d, e)
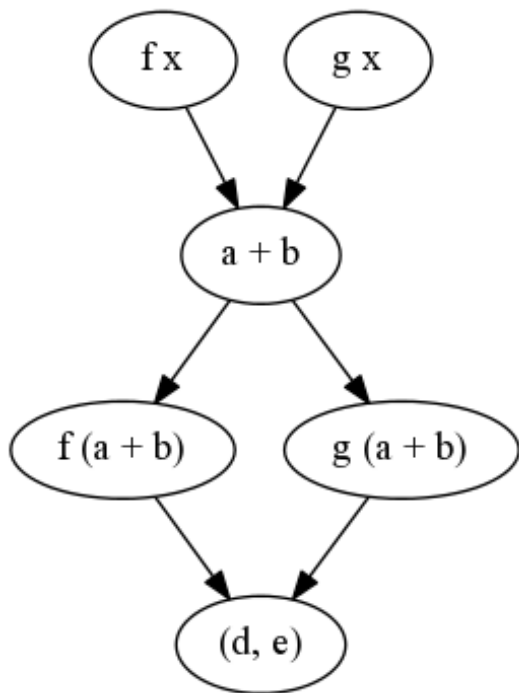
```haskell
import Control.Monad
import Control.Monad.Par

f, g :: Int -> Int
f x = x + 10
g x = x * 10

--    f x       g x
--      \      /
--        a + b
--        /   \
-- f (a+b)   g (a+b)
--        \   /
--        (d,e)

example1 :: Int -> (Int, Int)
example1 x = runPar $ do
  [a,b,c,d,e] <- replicateM 5 new
  fork (put a (f x))
  fork (put b (g x))
  a' <- get a
  b' <- get b
  fork (put c (a' + b'))
  c' <- get c
  fork (put d (f c'))
  fork (put e (g c'))
  d' <- get d
  e' <- get e
  return (d', e')

example2 :: [Int]
example2 = runPar $ do
  xs <- parMap (+1) [1..25]
  return xs
```

```haskell
-- foldr (+) 0 (map (^2) [1..xs])
example3 :: Int -> Int
example3 n = runPar $ do
  let range = (InclusiveRange 1 n)
  let mapper x = return (x^2)
  let reducer x y = return (x+y)
  parMapReduceRangeThresh 10 range mapper reducer 0
```

async

Async is a higher level set of functions that work on top of Control.Concurrent and STM.

```haskell
async :: IO a -> IO (Async a)
wait :: Async a -> IO a
cancel :: Async a -> IO ()
concurrently :: IO a -> IO b -> IO (a, b)
race :: IO a -> IO b -> IO (Either a b)
```

```haskell
import Control.Monad
import Control.Applicative
import Control.Concurrent
import Control.Concurrent.Async
import Data.Time

timeit :: IO a -> IO (a,Double)
timeit io = do
  t0 <- getCurrentTime
  a <- io
  t1 <- getCurrentTime
  return (a, realToFrac (t1 `diffUTCTime` t0))

worker :: Int -> IO Int
worker n = do
  -- simulate some work
  threadDelay (10^2 * n)
  return (n * n)

-- Spawn 2 threads in parallel, halt on both finished.
test1 :: IO (Int, Int)
test1 = do
  val1 <- async $ worker 1000
  val2 <- async $ worker 2000
  (,) <$> wait val1 <*> wait val2

-- Spawn 2 threads in parallel, halt on first finished.
test2 :: IO (Either Int Int)
test2 = do
  let val1 = worker 1000
```

```
    let val2 = worker 2000
    race val1 val2

-- Spawn 10000 threads in parallel, halt on all finished.
test3 :: IO [Int]
test3 = mapConcurrently worker [0..10000]


main :: IO ()
main = do
  print =<< timeit test1
  print =<< timeit test2
  print =<< timeit test3
```

Graphics

Diagrams

Diagrams is a a parser combinator library for generating vector images to SVG and a variety of other formats.

```
import Diagrams.Prelude
import Diagrams.Backend.SVG.CmdLine

sierpinski :: Int -> Diagram SVG R2
sierpinski 1 = eqTriangle 1
sierpinski n =
      s
    ===
  (s ||| s) # centerX
  where
    s = sierpinski (n - 1)

example :: Diagram SVG R2
example = sierpinski 5 # fc black


main :: IO ()
main = defaultMain example
```

```
$ runhaskell diagram1.hs -w 256 -h 256 -o diagram1.svg
```

See: Diagrams Quick Start Tutorial

Gloss

Parsing

Parsec

For parsing in Haskell it is quite common to use a family of libraries known as *Parser Combinators* which let us write code to generate parsers which themselves looks very similar to the parser grammar itself!

Combinators

`<|>` The choice operator tries to parse the first argument before proceeding to the second. Can be chained sequentially to a generate a sequence of options.

`many` Consumes an arbitrary number of patterns matching the given pattern and returns them as a list.

`many1` Like many but requires at least one match.

`optional` Optionally parses a given pattern returning its value as a Maybe.

`try` Backtracking operator will let us parse ambiguous matching expressions and restart with a different pattern.

There are two styles of writing Parsec, one can choose to write with monads or with applicatives.

```
parseM :: Parser Expr
parseM = do
  a <- identifier
  char '+'
  b <- identifier
  return $ Add a b
```

The same code written with applicatives uses the applicative combinators:

```
-- | Sequential application.
(<*>) :: f (a -> b) -> f a -> f b

-- | Sequence actions, discarding the value of the first argument.
(*>) :: f a -> f b -> f b
```

```
(*>) = liftA2 (const id)

-- | Sequence actions, discarding the value of the second argument.
(<*) :: f a -> f b -> f a
(<*) = liftA2 const



parseA :: Parser Expr
parseA = Add <$> identifier <* char '+' <*> identifier
```

Now for instance if we want to parse simple lambda expressions we can encode the parser logic as compositions of these combinators which yield the string parser when evaluated under with the `parse` .

```
import Text.Parsec
import Text.Parsec.String

data Expr
  = Var Char
  | Lam Char Expr
  | App Expr Expr
  deriving Show

lam :: Parser Expr
lam = do
  char '\\'
  n <- letter
  string "->"
  e <- expr
  return $ Lam n e

app :: Parser Expr
app = do
  apps <- many1 term
  return $ foldl1 App apps

var :: Parser Expr
var = do
  n <- letter
  return $ Var n

parens :: Parser Expr -> Parser Expr
parens p = do
  char '('
  e <- p
  char ')'
  return e

term :: Parser Expr
term = var <|> parens expr
```

```haskell
expr :: Parser Expr
expr = lam <|> app

decl :: Parser Expr
decl = do
  e <- expr
  eof
  return e

test :: IO ()
test = parseTest decl "\\y->y(\\x->x)y"

main :: IO ()
main = test >>= print
```

Custom Lexer

In our previous example lexing pass was not necessary because each lexeme mapped to a sequential collection of characters in the stream type. If we wanted to extend this parser with a non-trivial set of tokens, then Parsec provides us with a set of functions for defining lexers and integrating these with the parser combinators. The simplest example builds on top of the builtin Parsec language definitions which define a set of most common lexical schemes.

```haskell
haskellDef   :: LanguageDef st
emptyDef     :: LanguageDef st
haskellStyle :: LanguageDef st
javaStyle    :: LanguageDef st
```

For instance we'll build on top of the empty language grammar.

```haskell
import Text.Parsec
import Text.Parsec.Expr
import Text.Parsec.String
import qualified Text.Parsec.Token as Token

lexerStyle :: Token.LanguageDef ()
lexerStyle = Token.LanguageDef
  { Token.commentStart   = "{-"
  , Token.commentEnd     = "-}"
  , Token.commentLine    = "--"
  , Token.nestedComments = True
  , Token.identStart     = letter
  , Token.identLetter    = alphaNum <|> oneOf "_"
  , Token.opStart        = Token.opLetter lexerStyle
  , Token.opLetter       = oneOf "`~!@$%^&*-+=;:<>./?"
  , Token.reservedOpNames= []
  , Token.reservedNames  = ["if", "then", "else", "def"]
  , Token.caseSensitive  = True
  }
```

```haskell
lexer :: Token.TokenParser ()
lexer = Token.makeTokenParser lexerStyle

parens :: Parser a -> Parser a
parens = Token.parens lexer

natural :: Parser Integer
natural = Token.natural lexer

identifier :: Parser String
identifier = Token.identifier lexer

reservedOp :: String -> Parser ()
reservedOp = Token.reservedOp lexer

reserved :: String -> Parser ()
reserved = Token.reserved lexer

whiteSpace :: Parser ()
whiteSpace = Token.whiteSpace lexer

comma :: Parser String
comma = Token.comma lexer
```

See: Text.ParserCombinators.Parsec.Language

Simple Parsing

Putting our lexer and parser together we can write down a more robust parser for our little lambda calculus syntax.

```haskell
module Parser (parseExpr) where

import Text.Parsec
import Text.Parsec.String (Parser)
import Text.Parsec.Language (haskellStyle)

import qualified Text.Parsec.Expr as Ex
import qualified Text.Parsec.Token as Tok

type Id = String

data Expr
  = Lam Id Expr
  | App Expr Expr
  | Var Id
  | Num Int
  | Op  Binop Expr Expr
  deriving (Show)

data Binop = Add | Sub | Mul deriving Show
```

```haskell
lexer :: Tok.TokenParser ()
lexer = Tok.makeTokenParser style
  where ops = ["->","\\","+","*","-","="]
        style = haskellStyle {Tok.reservedOpNames = ops }

reservedOp :: String -> Parser ()
reservedOp = Tok.reservedOp lexer

identifier :: Parser String
identifier = Tok.identifier lexer

parens :: Parser a -> Parser a
parens = Tok.parens lexer

contents :: Parser a -> Parser a
contents p = do
  Tok.whiteSpace lexer
  r <- p
  eof
  return r

natural :: Parser Integer
natural = Tok.natural lexer

variable :: Parser Expr
variable = do
  x <- identifier
  return (Var x)

number :: Parser Expr
number = do
  n <- natural
  return (Num (fromIntegral n))

lambda :: Parser Expr
lambda = do
  reservedOp "\\"
  x <- identifier
  reservedOp "->"
  e <- expr
  return (Lam x e)

aexp :: Parser Expr
aexp =  parens expr
    <|> variable
    <|> number
    <|> lambda

term :: Parser Expr
term = Ex.buildExpressionParser table aexp
  where infixOp x f = Ex.Infix (reservedOp x >> return f)
        table = [[infixOp "*" (Op Mul) Ex.AssocLeft],
                 [infixOp "+" (Op Add) Ex.AssocLeft]]
```

```haskell
expr :: Parser Expr
expr = do
  es <- many1 term
  return (foldl1 App es)

parseExpr :: String -> Expr
parseExpr input =
  case parse (contents expr) "<stdin>" input of
    Left err -> error (show err)
    Right ast -> ast

main :: IO ()
main = getLine >>= print . parseExpr >> main
```

Trying it out:

```
λ: runhaskell simpleparser.hs
1+2
Op Add (Num 1) (Num 2)

\i -> \x -> x
Lam "i" (Lam "x" (Var "x"))

\s -> \f -> \g -> \x -> f x (g x)
Lam "s" (Lam "f" (Lam "g" (Lam "x" (App (App (Var "f") (Var "x")) (App (Var "g") (Var
```

Stateful Parsing

For a more complex use, consider parser that are internally stateful, for example adding operators that can defined at parse-time and are dynamically added to the `expressionParser` table upon definition.

```haskell
module Main where

import qualified Text.Parsec.Expr as Ex
import qualified Text.Parsec.Token as Tok

import Text.Parsec.Language (haskellStyle)

import Data.List
import Data.Function

import Control.Monad.Identity (Identity)

import Text.Parsec
import qualified Text.Parsec as P

type Name = String
```

```haskell
data Expr
  = Var Name
  | Lam Name Expr
  | App Expr Expr
  | Let Name Expr Expr
  | BinOp Name Expr Expr
  | UnOp Name Expr
  deriving (Show)

data Assoc
  = OpLeft
  | OpRight
  | OpNone
  | OpPrefix
  | OpPostfix
  deriving Show

data Decl
  = LetDecl Expr
  | OpDecl OperatorDef
  deriving (Show)

type Op x = Ex.Operator String ParseState Identity x
type Parser a = Parsec String ParseState a
data ParseState = ParseState [OperatorDef] deriving Show

data OperatorDef = OperatorDef {
    oassoc :: Assoc
  , oprec :: Integer
  , otok :: Name
  } deriving Show

lexer :: Tok.GenTokenParser String u Identity
lexer = Tok.makeTokenParser style
  where ops = ["->","\\","+","*","<","=","[","]","_"]
        names = ["let","in","infixl", "infixr", "infix", "postfix", "prefix"]
        style = haskellStyle { Tok.reservedOpNames = ops
                             , Tok.reservedNames = names
                             , Tok.identLetter = alphaNum <|> oneOf "#'_"
                             , Tok.commentLine = "--"
                             }

reserved   = Tok.reserved lexer
reservedOp = Tok.reservedOp lexer
identifier = Tok.identifier lexer
parens     = Tok.parens lexer
brackets   = Tok.brackets lexer
braces     = Tok.braces lexer
commaSep   = Tok.commaSep lexer
semi       = Tok.semi lexer
integer    = Tok.integer lexer
chr        = Tok.charLiteral lexer
str        = Tok.stringLiteral lexer
operator   = Tok.operator lexer
```

```haskell
contents :: Parser a -> Parser a
contents p = do
  Tok.whiteSpace lexer
  r <- p
  eof
  return r

expr :: Parser Expr
expr = do
  es <- many1 term
  return (foldl1 App es)

lambda :: Parser Expr
lambda = do
  reservedOp "\\"
  args <- identifier
  reservedOp "->"
  body <- expr
  return $ Lam args body

letin :: Parser Expr
letin = do
  reserved "let"
  x <- identifier
  reservedOp "="
  e1 <- expr
  reserved "in"
  e2 <- expr
  return (Let x e1 e2)

variable :: Parser Expr
variable = do
  x <- identifier
  return (Var x)


addOperator :: OperatorDef -> Parser ()
addOperator a = P.modifyState $ \(ParseState ops) -> ParseState (a : ops)

mkTable :: ParseState -> [[Op Expr]]
mkTable (ParseState ops) =
  map (map toParser) $
    groupBy ((==) `on` oprec) $
      reverse $ sortBy (compare `on` oprec) $ ops

toParser :: OperatorDef -> Op Expr
toParser (OperatorDef ass _ tok) = case ass of
    OpLeft    -> infixOp tok (BinOp tok) (toAssoc ass)
    OpRight   -> infixOp tok (BinOp tok) (toAssoc ass)
    OpNone    -> infixOp tok (BinOp tok) (toAssoc ass)
    OpPrefix  -> prefixOp tok (UnOp tok)
    OpPostfix -> postfixOp tok (UnOp tok)
  where
```

```
    toAssoc OpLeft = Ex.AssocLeft
    toAssoc OpRight = Ex.AssocRight
    toAssoc OpNone = Ex.AssocNone
    toAssoc _ = error "no associativity"

infixOp :: String -> (a -> a -> a) -> Ex.Assoc -> Op a
infixOp x f = Ex.Infix (reservedOp x >> return f)

prefixOp :: String -> (a -> a) -> Ex.Operator String u Identity a
prefixOp name f = Ex.Prefix (reservedOp name >> return f)

postfixOp :: String -> (a -> a) -> Ex.Operator String u Identity a
postfixOp name f = Ex.Postfix (reservedOp name >> return f)

term :: Parser Expr
term = do
  tbl <- getState
  let table = mkTable tbl
  Ex.buildExpressionParser table aexp

aexp :: Parser Expr
aexp =  letin
    <|> lambda
    <|> variable
    <|> parens expr

letdecl :: Parser Decl
letdecl = do
  e <- expr
  return $ LetDecl e


opleft :: Parser Decl
opleft = do
  reserved "infixl"
  prec <- integer
  sym <- parens operator
  let op = (OperatorDef OpLeft prec sym)
  addOperator op
  return $ OpDecl op

opright :: Parser Decl
opright = do
  reserved "infixr"
  prec <- integer
  sym <- parens operator
  let op = (OperatorDef OpRight prec sym)
  addOperator op
  return $ OpDecl op

opnone :: Parser Decl
opnone = do
  reserved "infix"
  prec <- integer
```

```haskell
    sym <- parens operator
    let op = (OperatorDef OpNone prec sym)
    addOperator op
    return $ OpDecl op

opprefix :: Parser Decl
opprefix = do
  reserved "prefix"
  prec <- integer
  sym <- parens operator
  let op = OperatorDef OpPrefix prec sym
  addOperator op
  return $ OpDecl op

oppostfix :: Parser Decl
oppostfix = do
  reserved "postfix"
  prec <- integer
  sym <- parens operator
  let op = OperatorDef OpPostfix prec sym
  addOperator op
  return $ OpDecl op

decl :: Parser Decl
decl =
    try letdecl
    <|> opleft
    <|> opright
    <|> opnone
    <|> opprefix
    <|> oppostfix

top :: Parser Decl
top = do
  x <- decl
  P.optional semi
  return x


modl :: Parser [Decl]
modl = many top

parseModule :: SourceName -> String -> Either ParseError [Decl]
parseModule filePath = P.runParser (contents modl) (ParseState []) filePath

main :: IO ()
main = do
  input <- readFile "test.in"
  let res = parseModule "<stdin>" input
  case res of
    Left err -> print err
    Right ast -> mapM_ print ast
```

For example input try:

```
infixl 3 ($);
infixr 4 (#);

infix 4 (.);

prefix 10 (-);
postfix 10 (!);

let z = y in a $ a $ (-a)!;
let z = y in a # a # a $ b; let z = y in a # a # a # b;
```

Generic Parsing

Previously we defined generic operations for pretty printing and this begs the question of whether we can write a parser on top of Generics. The answer is generally yes, so long as there is a direct mapping between the specific lexemes and sum and products types. Consider the simplest case where we just read off the names of the constructors using the regular Generics machinery and then build a Parsec parser terms of them.

```
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE ScopedTypeVariables #-}

import Text.Parsec
import Text.Parsec.Text.Lazy
import Control.Applicative ((<*), (<*>), (<$>))
import GHC.Generics

class GParse f where
  gParse :: Parser (f a)

-- Type synonym metadata for constructors
instance (GParse f, Constructor c) => GParse (C1 c f) where
  gParse =
    let con = conName (undefined :: t c f a) in
    (fmap M1 gParse) <* string con

-- Constructor names
instance GParse f => GParse (D1 c f) where
  gParse = fmap M1 gParse

-- XXX
{-instance GParse f => GParse (S1 c f) where-}
  {-gParse = fmap M1 gParse-}

-- Sum types
```

```haskell
instance (GParse a, GParse b) => GParse (a :+: b) where
  gParse = try (fmap L1 gParse <|> fmap R1 gParse)

-- Product types
instance (GParse f, GParse g) => GParse (f :*: g) where
  gParse = (:*:) <$> gParse <*> gParse

-- Nullary constructors
instance GParse U1 where
  gParse = return U1

data Scientist
  = Newton
  | Einstein
  | Schrodinger
  | Feynman
  deriving (Show, Generic)

data Musician
  = Vivaldi
  | Bach
  | Mozart
  | Beethoven
  deriving (Show, Generic)

gparse :: (Generic g, GParse (Rep g)) => Parser g
gparse = fmap to gParse

scientist :: Parser Scientist
scientist = gparse

musician :: Parser Musician
musician = gparse
```

```
λ: parseTest parseMusician "Bach"
Bach

λ: parseTest parseScientist "Feynman"
Feynman
```

Attoparsec

Attoparsec is a parser combinator like Parsec but more suited for bulk parsing of large text and binary files instead of parsing language syntax to ASTs. When written properly Attoparsec parsers can be extremely efficient.

One notable distinction between Parsec and Attoparsec is that backtracking operator (`try`) is not present and reflects on attoparsec's different underlying parser model.

For a simple little lambda calculus language we can use attoparsec much in the same we used parsec:

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# OPTIONS_GHC -fno-warn-unused-do-bind #-}

import Control.Applicative
import Data.Attoparsec.Text
import qualified Data.Text as T
import qualified Data.Text.IO as T
import Data.List (foldl1')

data Name
  = Gen Int
  | Name T.Text
  deriving (Eq, Show, Ord)

data Expr
  = Var Name
  | App Expr Expr
  | Lam [Name] Expr
  | Lit Int
  | Prim PrimOp
  deriving (Eq, Show)

data PrimOp
  = Add
  | Sub
  | Mul
  | Div
  deriving (Eq, Show)

data Defn = Defn Name Expr
  deriving (Eq, Show)

name :: Parser Name
name = Name . T.pack <$> many1 letter

num :: Parser Expr
num = Lit <$> signed decimal

var :: Parser Expr
var = Var <$> name

lam :: Parser Expr
lam = do
  string "\\"
  vars <- many1 (skipSpace *> name)
  skipSpace *> string "->"
  body <- expr
  return (Lam vars body)

eparen :: Parser Expr
eparen = char '(' *> expr <* skipSpace <* char ')'
```

```
prim :: Parser Expr
prim = Prim <$> (
        char '+' *> return Add
  <|> char '-' *> return Sub
  <|> char '*' *> return Mul
  <|> char '/' *> return Div)

expr :: Parser Expr
expr = foldl1' App <$> many1 (skipSpace *> atom)

atom :: Parser Expr
atom = try lam
     <|> eparen
     <|> prim
     <|> var
     <|> num

def :: Parser Defn
def = do
  skipSpace
  nm <- name
  skipSpace *> char '=' *> skipSpace
  ex <- expr
  skipSpace <* char ';'
  return $ Defn nm ex

file :: T.Text -> Either String [Defn]
file = parseOnly (many def <* skipSpace)

parseFile :: FilePath -> IO (Either T.Text [Defn])
parseFile path = do
  contents <- T.readFile path
  case file contents of
    Left a -> return $ Left (T.pack a)
    Right b -> return $ Right b

main :: IO (Either T.Text [Defn])
main = parseFile "simple.ml"
```

For an example try the above parser with the following simple lambda expression.

```
f = g (x - 1);
g = f (x + 1);
h = \x y -> (f x) + (g y);
```

Attoparsec adapts very well to binary and network protocol style parsing as well, this is extracted from a small implementation of a distributed consensus network protocol:

```
{-# LANGUAGE OverloadedStrings #-}
```

```
import Control.Monad

import Data.Attoparsec
import Data.Attoparsec.Char8 as A
import Data.ByteString.Char8

data Action
  = Success
  | KeepAlive
  | NoResource
  | Hangup
  | NewLeader
  | Election
  deriving Show

type Sender = ByteString
type Payload = ByteString

data Message = Message
  { action :: Action
  , sender :: Sender
  , payload :: Payload
  } deriving Show

proto :: Parser Message
proto = do
  act  <- paction
  send <- A.takeTill (== '.')
  body <- A.takeTill (A.isSpace)
  endOfLine
  return $ Message act send body

paction :: Parser Action
paction = do
  c <- anyWord8
  case c of
    1  -> return Success
    2  -> return KeepAlive
    3  -> return NoResource
    4  -> return Hangup
    5  -> return NewLeader
    6  -> return Election
    _  -> mzero

main :: IO ()
main = do
  let msgtext = "\x01\x6c\x61\x70\x74\x6f\x70\x2e\x33\x2e\x31\x34\x31\x35\x39\x32\x36\
  let msg = parseOnly proto msgtext
  print msg
```

See: Text Parsing Tutorial

Streaming

Lazy IO

The problem with using the usual monadic approach to processing data accumulated through IO is that the Prelude tools require us to manifest large amounts of data in memory all at once before we can even begin computation.

```haskell
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
sequence :: Monad m => [m a] -> m [a]
```

Reading from the file creates a thunk for the string that forced will then read the file. The problem is then that this method ties the ordering of IO effects to evaluation order which is difficult to reason about in the large.

Consider that normally the monad laws (in the absence of `seq`) guarantee that these computations should be identical. But using lazy IO we can construct a degenerate case.

```haskell
import System.IO

main :: IO ()
main = do
  withFile "foo.txt" ReadMode $ \fd -> do
    contents <- hGetContents fd
    print contents
  -- "foo\n"

  contents <- withFile "foo.txt" ReadMode hGetContents
  print contents
  -- ""
```

So what we need is a system to guarantee deterministic resource handling with constant memory usage. To that end both the Conduits and Pipes libraries solved this problem using different (though largely equivalent) approaches.

Pipes

```haskell
await :: Monad m => Pipe a y m a
yield :: Monad m => a -> Pipe x a m ()

(>->) :: Monad m
      => Pipe a b m r
      -> Pipe b c m r
      -> Pipe a c m r

runEffect :: Monad m => Effect m r -> m r
toListM :: Monad m => Producer a m () -> m [a]
```

Pipes is a stream processing library with a strong emphasis on the static semantics of composition. The simplest usage is to connect "pipe" functions with a `(>->)` composition operator, where each component can `await` and `yield` to push and pull values along the stream.

```haskell
import Pipes
import Pipes.Prelude as P
import Control.Monad
import Control.Monad.Identity

a :: Producer Int Identity ()
a = forM_ [1..10] yield

b :: Pipe Int Int Identity ()
b =  forever $ do
  x <- await
  yield (x*2)
  yield (x*3)
  yield (x*4)

c :: Pipe Int Int Identity ()
c = forever $ do
  x <- await
  if (x `mod` 2) == 0
    then yield x
    else return ()

result :: [Int]
result = P.toList $ a >-> b >-> c
```

For example we could construct a "FizzBuzz" pipe.

```haskell
{-# LANGUAGE MultiWayIf #-}

import Pipes
import qualified Pipes.Prelude as P

count :: Producer Integer IO ()
count = each [1..100]

fizzbuzz :: Pipe Integer String IO ()
fizzbuzz = do
  n <- await
  if | n `mod` 15 == 0 -> yield "FizzBuzz"
     | n `mod` 5  == 0 -> yield "Fizz"
     | n `mod` 3  == 0 -> yield "Buzz"
     | otherwise       -> return ()
  fizzbuzz

main :: IO ()
main = runEffect $ count >-> fizzbuzz >-> P.stdoutLn
```

To continue with the degenerate case we constructed with Lazy IO, consider than we can now compose and sequence deterministic actions over files without having to worry about effect order.

```haskell
import Pipes
import Pipes.Prelude as P
import System.IO

readF :: FilePath -> Producer String IO ()
readF file = do
    lift $ putStrLn $ "Opened" ++ file
    h <- lift $ openFile file ReadMode
    fromHandle h
    lift $ putStrLn $ "Closed" ++ file
    lift $ hClose h

main :: IO ()
main = runEffect $ readF "foo.txt" >-> P.take 3 >-> stdoutLn
```

This is simple a sampling of the functionality of lens. The documentation for pipes is extensive and great deal of care has been taken make the library extremely thorough. `pipes` is a shining example of an accessible yet category theoretic driven design.

See: Pipes Tutorial

Safe Pipes

```haskell
bracket :: MonadSafe m => Base m a -> (a -> Base m b) -> (a -> m c) -> m c
```

As a motivating example, ZeroMQ is a network messaging library that abstracts over traditional Unix sockets to a variety of network topologies. Most notably it isn't designed to guarantee any sort of transactional guarantees for delivery or recovery in case of errors so it's necessary to design a layer on top of it to provide the desired behavior at the application layer.

In Haskell we'd like to guarantee that if we're polling on a socket we get messages delivered in a timely fashion or consider the resource in an error state and recover from it. Using `pipes-safe` we can manage the life cycle of lazy IO resources and can safely handle failures, resource termination and finalization gracefully. In other languages this kind of logic would be smeared across several places, or put in some global context and prone to introduce errors and subtle race conditions. Using pipes we instead get a nice tight abstraction designed exactly to fit this kind of use case.

For instance now we can bracket the ZeroMQ socket creation and finalization within the `SafeT` monad transformer which guarantees that after successful message delivery we execute the pipes function as expected, or on failure we halt the execution and finalize the socket.

```haskell
import Pipes
import Pipes.Safe
import qualified Pipes.Prelude as P

import System.Timeout (timeout)
import Data.ByteString.Char8
```

```haskell
import qualified System.ZMQ as ZMQ

data Opts = Opts
  { _addr    :: String  -- ^ ZMQ socket address
  , _timeout :: Int     -- ^ Time in milliseconds for socket timeout
  }

recvTimeout :: Opts -> ZMQ.Socket a -> Producer ByteString (SafeT IO) ()
recvTimeout opts sock = do
  body <- liftIO $ timeout (_timeout opts) (ZMQ.receive sock [])
  case body of
    Just msg -> do
      liftIO $ ZMQ.send sock msg []
      yield msg
      recvTimeout opts sock
    Nothing  -> liftIO $ print "socket timed out"

collect :: ZMQ.Context
        -> Opts
        -> Producer ByteString (SafeT IO) ()
collect ctx opts = bracket zinit zclose (recvTimeout opts)
  where
    -- Initialize the socket
    zinit = do
      liftIO $ print "waiting for messages"
      sock <- ZMQ.socket ctx ZMQ.Rep
      ZMQ.bind sock (_addr opts)
      return sock

    -- On timeout or completion guarantee the socket get closed.
    zclose sock = do
      liftIO $ print "finalizing"
      ZMQ.close sock

runZmq :: ZMQ.Context -> Opts -> IO ()
runZmq ctx opts = runSafeT $ runEffect $
  collect ctx opts >-> P.take 10 >-> P.print

main :: IO ()
main = do
  ctx <- ZMQ.init 1
  let opts = Opts {_addr = "tcp://127.0.0.1:8000", _timeout = 1000000 }
  runZmq ctx opts
  ZMQ.term ctx
```

Conduits

```haskell
await :: Monad m => ConduitM i o m (Maybe i)
yield :: Monad m => o -> ConduitM i o m ()
($$) :: Monad m => Source m a -> Sink a m b -> m b
(=$) :: Monad m => Conduit a m b -> Sink b m c -> Sink a m c
```

```haskell
type Sink i = ConduitM i Void
type Source m o = ConduitM () o m ()
type Conduit i m o = ConduitM i o m ()
```

Conduits are conceptually similar though philosophically different approach to the same problem of constant space deterministic resource handling for IO resources.

The first initial difference is that await function now returns a `Maybe` which allows different handling of termination. The composition operators are also split into a connecting operator (`$$`) and a fusing operator (`=$`) for combining Sources and Sink and a Conduit and a Sink respectively.

```haskell
{-# LANGUAGE MultiWayIf #-}

import Data.Conduit
import Control.Monad.Trans
import qualified Data.Conduit.List as CL

source :: Source IO Int
source = CL.sourceList [1..100]

conduit :: Conduit Int IO String
conduit = do
  val <- await
  liftIO $ print val
  case val of
    Nothing -> return ()
    Just n -> do
      if | n `mod` 15 == 0 -> yield "FizzBuzz"
         | n `mod` 5  == 0 -> yield "Fizz"
         | n `mod` 3  == 0 -> yield "Buzz"
         | otherwise       -> return ()
      conduit

sink :: Sink String IO ()
sink = CL.mapM_ putStrLn

main :: IO ()
main = source $$ conduit =$ sink
```

See: Conduit Overview

Data Formats

JSON

Aeson is library for efficient parsing and generating JSON.

```haskell
decode :: FromJSON a => ByteString -> Maybe a
encode :: ToJSON a => a -> ByteString
```

```
eitherDecode :: FromJSON a => ByteString -> Either String a

fromJSON :: FromJSON a => Value -> Result a
toJSON :: ToJSON a => a -> Value
```

We'll work with this contrived example:

```
{
    "id": 1,
    "name": "A green door",
    "price": 12.50,
    "tags": ["home", "green"],
    "refs": {
      "a": "red",
      "b": "blue"
    }
}
```

Aeson uses several high performance data structures (Vector, Text, HashMap) by default instead of the naive versions so typically using Aeson will require that us import them and use `OverloadedStrings` when indexing into objects.

```
type Object = HashMap Text Value

type Array = Vector Value

-- | A JSON value represented as a Haskell value.
data Value = Object !Object
           | Array !Array
           | String !Text
           | Number !Scientific
           | Bool !Bool
           | Null
```

See: [Aeson Documentation](#)

Unstructured

In dynamic scripting languages it's common to parse amorphous blobs of JSON without any a priori structure and then handle validation problems by throwing exceptions while traversing it. We can do the same using Aeson and the Maybe monad.

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Text
import Data.Aeson
import Data.Vector
import qualified Data.HashMap.Strict as M
import qualified Data.ByteString.Lazy as BL
```

```haskell
-- Pull a key out of an JSON object.
(^?) :: Value -> Text -> Maybe Value
(^?) (Object obj) k = M.lookup k obj
(^?) _ _ = Nothing

-- Pull the ith value out of a JSON list.
ix :: Value -> Int -> Maybe Value
ix (Array arr) i = arr !? i
ix _ _ = Nothing

readJSON str = do
  obj   <- decode str
  price <- obj ^? "price"
  refs  <- obj ^? "refs"
  tags  <- obj ^? "tags"
  aref  <- refs ^? "a"
  tag1  <- tags `ix` 0
  return (price, aref, tag1)

main :: IO ()
main = do
  contents <- BL.readFile "example.json"
  print $ readJSON contents
```

Structured

This isn't ideal since we've just smeared all the validation logic across our traversal logic instead of separating concerns and handling validation in separate logic. We'd like to describe the structure before-hand and the invalid case separately. Using Generic also allows Haskell to automatically write the serializer and deserializer between our datatype and the JSON string based on the names of record field names.

```haskell
{-# LANGUAGE DeriveGeneric #-}

import Data.Text
import Data.Aeson
import GHC.Generics
import qualified Data.ByteString.Lazy as BL

import Control.Applicative

data Refs = Refs
  { a :: String
  , b :: String
  } deriving (Show,Generic)

data Data = Data
  { id    :: Int
  , name  :: Text
  , price :: Int
  , tags  :: [String]
  , refs  :: Refs
```

```
  } deriving (Show,Generic)

instance FromJSON Data
instance FromJSON Refs
instance ToJSON Data
instance ToJSON Refs

main :: IO ()
main = do
  contents <- BL.readFile "example.json"
  let Just dat = decode contents
  print $ name dat
  print $ a (refs dat)
```

Now we get our validated JSON wrapped up into a nicely typed Haskell ADT.

```
Data
  { id = 1
  , name = "A green door"
  , price = 12
  , tags = [ "home" , "green" ]
  , refs = Refs { a = "red" , b = "blue" }
  }
```

The functions `fromJSON` and `toJSON` can be used to convert between this sum type and regular Haskell types with.

```
data Result a = Error String | Success a
```

```
λ: fromJSON (Bool True) :: Result Bool
Success True

λ: fromJSON (Bool True) :: Result Double
Error "when expecting a Double, encountered Boolean instead"
```

CSV

Cassava is an efficient CSV parser library. We'll work with this tiny snippet from the iris dataset:

```
sepal_length,sepal_width,petal_length,petal_width,plant_class
5.1,3.5,1.4,0.2,Iris-setosa
5.0,2.0,3.5,1.0,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
```

Unstructured

Just like with Aeson if we really want to work with unstructured data the library accommodates this.

```haskell
import Data.Csv

import Text.Show.Pretty

import qualified Data.Vector as V
import qualified Data.ByteString.Lazy as BL

type ErrorMsg = String
type CsvData = V.Vector (V.Vector BL.ByteString)

example :: FilePath -> IO (Either ErrorMsg CsvData)
example fname = do
  contents <- BL.readFile fname
  return $ decode NoHeader contents
```

We see we get the nested set of stringy vectors:

```
[ [ "sepal_length"
  , "sepal_width"
  , "petal_length"
  , "petal_width"
  , "plant_class"
  ]
, [ "5.1" , "3.5" , "1.4" , "0.2" , "Iris-setosa" ]
, [ "5.0" , "2.0" , "3.5" , "1.0" , "Iris-versicolor" ]
, [ "6.3" , "3.3" , "6.0" , "2.5" , "Iris-virginica" ]
]
```

Structured

Just like with Aeson we can use Generic to automatically write the deserializer between our CSV data and our custom datatype.

```haskell
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE DeriveGeneric #-}

import Data.Csv
import GHC.Generics
import qualified Data.Vector as V
import qualified Data.ByteString.Lazy as BL

data Plant = Plant
  { sepal_length :: Double
  , sepal_width  :: Double
  , petal_length :: Double
```

```haskell
    , petal_width  :: Double
    , plant_class :: String
    } deriving (Generic, Show)

instance FromNamedRecord Plant
instance ToNamedRecord Plant

type ErrorMsg = String
type CsvData = (Header, V.Vector Plant)

parseCSV :: FilePath -> IO (Either ErrorMsg CsvData)
parseCSV fname = do
  contents <- BL.readFile fname
  return $ decodeByName contents

main = parseCSV "iris.csv" >>= print
```

And again we get a nice typed ADT as a result.

```
[ Plant
    { sepal_length = 5.1
    , sepal_width = 3.5
    , petal_length = 1.4
    , petal_width = 0.2
    , plant_class = "Iris-setosa"
    }
, Plant
    { sepal_length = 5.0
    , sepal_width = 2.0
    , petal_length = 3.5
    , petal_width = 1.0
    , plant_class = "Iris-versicolor"
    }
, Plant
    { sepal_length = 6.3
    , sepal_width = 3.3
    , petal_length = 6.0
    , petal_width = 2.5
    , plant_class = "Iris-virginica"
    }
]
```

Network & Web Programming

HTTP

Haskell has a variety of HTTP request and processing libraries.

```haskell
{-# LANGUAGE OverloadedStrings #-}
```

```haskell
import Network.HTTP.Types
import Network.HTTP.Client
import Control.Applicative
import Control.Concurrent.Async

type URL = String

get :: Manager -> URL -> IO Int
get m url = do
  req <- parseUrl url
  statusCode <$> responseStatus <$> httpNoBody req m

single :: IO Int
single = do
  withManager defaultManagerSettings $ \m -> do
    get m "http://haskell.org"

parallel :: IO [Int]
parallel = do
  withManager defaultManagerSettings $ \m -> do
    -- Fetch w3.org 10 times concurrently
    let urls = replicate 10 "http://www.w3.org"
    mapConcurrently (get m) urls

main :: IO ()
main = do
  print =<< single
  print =<< parallel
```

Warp

Warp is a particularly efficient web server, it's the backed request engine behind several of popular Haskell web frameworks. The internals have been finely tuned to utilize Haskell's concurrent runtime and is capable of handling a great deal of concurrent requests.

```haskell
{-# LANGUAGE OverloadedStrings #-}

import Network.Wai
import Network.Wai.Handler.Warp (run)
import Network.HTTP.Types

app :: Application
app req = return $ responseLBS status200 [] "Engage!"

main :: IO ()
main = run 8000 app
```

See: Warp

Scotty

Continuing with our trek through web libraries, Scotty is a web microframework similar in principle to Flask in Python or Sinatra in Ruby.

```haskell
{-# LANGUAGE OverloadedStrings #-}

import Web.Scotty

import qualified Text.Blaze.Html5 as H
import Text.Blaze.Html5 (toHtml, Html)
import Text.Blaze.Html.Renderer.Text (renderHtml)

greet :: String -> Html
greet user = H.html $ do
  H.head $
    H.title "Welcome!"
  H.body $ do
    H.h1 "Greetings!"
    H.p ("Hello " >> toHtml user >> "!")

app = do
  get "/" $
    text "Home Page"

  get "/greet/:name" $ do
    name <- param "name"
    html $ renderHtml (greet name)

main :: IO ()
main = scotty 8000 app
```

Of importance to note is the Blaze library used here overloads do-notation but is not itself a proper monad so the various laws and invariants that normally apply for monads may break down or fail with error terms.

See: Making a Website with Haskell

Databases

Acid State

Acid-state allows us to build a "database on demand" for arbitrary Haskell datatypes that guarantees atomic transactions. For example, we can build a simple key-value store wrapped around the Map type.

```haskell
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE TemplateHaskell #-}

import Data.Acid
import Data.Typeable
```

```haskell
import Data.SafeCopy
import Control.Monad.Reader (ask)

import qualified Data.Map as Map
import qualified Control.Monad.State as S

type Key = String
type Value = String

data Database = Database !(Map.Map Key Value)
    deriving (Show, Ord, Eq, Typeable)

$(deriveSafeCopy 0 'base ''Database)

insertKey :: Key -> Value -> Update Database ()
insertKey key value
    = do Database m <- S.get
         S.put (Database (Map.insert key value m))

lookupKey :: Key -> Query Database (Maybe Value)
lookupKey key
    = do Database m <- ask
         return (Map.lookup key m)

deleteKey :: Key -> Update Database ()
deleteKey key
    = do Database m <- S.get
         S.put (Database (Map.delete key m))

allKeys :: Int -> Query Database [(Key, Value)]
allKeys limit
    = do Database m <- ask
         return $ take limit (Map.toList m)

$(makeAcidic ''Database ['insertKey, 'lookupKey, 'allKeys, 'deleteKey])

fixtures :: Map.Map String String
fixtures = Map.empty

test ::  Key -> Value -> IO ()
test key val = do
    database <- openLocalStateFrom "db/" (Database fixtures)
    result <- update database (InsertKey key val)
    result <- query database (AllKeys 10)
    print result
```
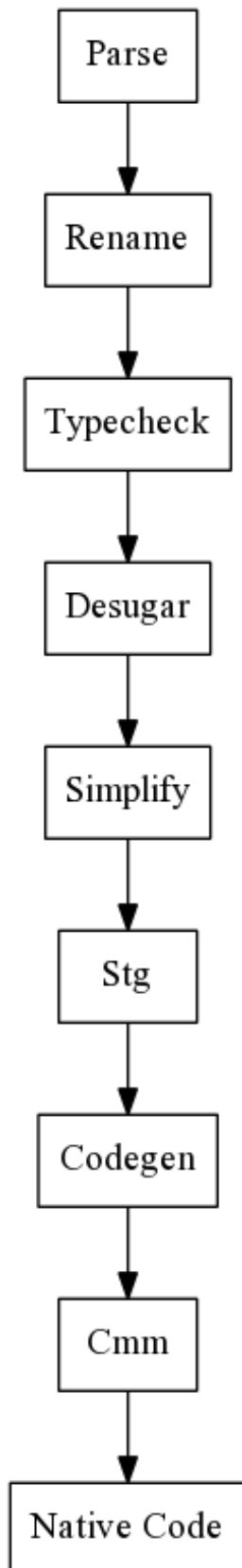
GHC

Block Diagram

The flow of code through GHC is a process of translation between several intermediate languages and optimizations and transformations thereof. A common pattern for many of these AST types is they are parametrized over a binder type and at various stages the binders will be transformed, for example the

Renamer pass effectively translates the `HsSyn` datatype from a AST parametrized over literal strings as the user enters into a `HsSyn` parameterized over qualified names that includes modules and package names into a higher level Name type.

```
┌──────────────┐
│    Parse     │
└──────────────┘
        │
        ▼
┌──────────────┐
│    Rename    │
└──────────────┘
        │
        ▼
┌──────────────┐
│  Typecheck   │
└──────────────┘
        │
        ▼
┌──────────────┐
│   Desugar    │
└──────────────┘
        │
        ▼
┌──────────────┐
│   Simplify   │
└──────────────┘
        │
        ▼
┌──────────────┐
│     Stg      │
└──────────────┘
        │
        ▼
┌──────────────┐
│   Codegen    │
└──────────────┘
        │
        ▼
┌──────────────┐
│     Cmm      │
└──────────────┘
        │
        ▼
┌──────────────┐
│ Native Code  │
└──────────────┘
```

- **Parser/Frontend**: An enormous AST translated from human syntax that makes explicit possible all expressible syntax (declarations, do-notation, where clauses, syntax extensions, template haskell, ...). This is unfiltered Haskell and it is *enormous*.
- **Renamer** takes syntax from the frontend and transforms all names to be qualified (`base:Prelude.map` instead of `map`) and any shadowed names in lambda binders transformed into unique names.
- **Typechecker** is a large pass that serves two purposes, first is the core type bidirectional

inference engine where most of the work happens and the translation between the frontend `Core` syntax.

- **Desugarer** translates several higher level syntactic constructors
    - `where` statements are turned into (possibly recursive) nested `let` statements.
    - Nested pattern matches are expanded out into splitting trees of case statements.
    - do-notation is expanded into explicit bind statements.
    - Lots of others.
- **Simplifier** transforms many Core constructs into forms that are more adaptable to compilation. For example let statements will be floated or raised, pattern matches will simplified, inner loops will be pulled out and transformed into more optimal forms. Non-intuitively the resulting may actually be much more complex (for humans) after going through the simplifier!
- **Stg** pass translates the resulting Core into STG (Spineless Tagless G-Machine) which effectively makes all laziness explicit and encodes the thunks and update frames that will be handled during evaluation.
- **Codegen/Cmm** pass will then translate STG into Cmm (flavoured C--) a simple imperative language that manifests the low-level implementation details of runtime types. The runtime closure types and stack frames are made explicit and low-level information about the data and code (arity, updatability, free variables, pointer layout) made manifest in the info tables present on most constructs.
- **Native Code** The final pass will than translate the resulting code into either LLVM or Assembly via either through GHC's home built native code generator (NCG) or the LLVM backend.

Information for about each pass can dumped out via a rather large collection of flags. The GHC internals are very accessible although some passes are somewhat easier to understand than others. Most of the time `-ddump-simpl` and `-ddump-stg` are sufficient to get an understanding of how the code will compile, unless of course you're dealing with very specialized optimizations or hacking on GHC itself.

Flag	Action
`-ddump-parsed`	Frontend AST.
`-ddump-rn`	Output of the rename pass.
`-ddump-tc`	Output of the typechecker.
`-ddump-splices`	Output of TemplateHaskell splices.
`-ddump-types`	Typed AST representation.
`-ddump-deriv`	Output of deriving instances.
`-ddump-ds`	Output of the desugar pass.
`-ddump-spec`	Output of specialisation pass.
`-ddump-rules`	Output of applying rewrite rules.
`-ddump-vect`	Output results of vectorize pass.
`-ddump-simpl`	Ouptut of the SimplCore pass.
`-ddump-inlinings`	Output of the inliner.
`-ddump-cse`	Output of the common subexpression elimination pass.
`-ddump-prep`	The CorePrep pass.
`-ddump-stg`	The resulting STG.
`-ddump-cmm`	The resulting Cmm.
`-ddump-opt-cmm`	The resulting Cmm optimization pass.
`-ddump-asm`	The final assembly generated.
`-ddump-llvm`	The final LLVM IR generated.

Core

Core is the explicitly typed System-F family syntax through that all Haskell constructs can be expressed in.

To inspect the core from GHCi we can invoke it using the following flags and the following shell alias. We have explicitly disable the printing of certain metadata and longform names to make the representation easier to read.

```
alias ghci-core="ghci -ddump-simpl -dsuppress-idinfo \
-dsuppress-coercions -dsuppress-type-applications \
-dsuppress-uniques -dsuppress-module-prefixes"
```

At the interactive prompt we can then explore the core representation interactively:

```
$ ghci-core
λ: let f x = x + 2 ; f :: Int -> Int

==================== Simplified expression ====================
returnIO
  (: ((\ (x :: Int) -> + $fNumInt x (I# 2)) `cast` ...) ([]))

λ: let f x = (x, x)

==================== Simplified expression ====================
returnIO (: ((\ (@ t) (x :: t) -> (x, x)) `cast` ...) ([]))
```

ghc-core is also very useful for looking at GHC's compilation artifacts.

```
$ ghc-core --no-cast --no-asm
```

Alternatively the major stages of the compiler (parse tree, core, stg, cmm, asm) can be manually outputted and inspected by passing several flags to the compiler:

```
$ ghc -ddump-to-file -ddump-parsed -ddump-simpl -ddump-stg -ddump-cmm -ddump-asm
```

Reading Core

Core from GHC is roughly human readable, but it's helpful to look at simple human written examples to get the hang of what's going on.

```
id :: a -> a
id x = x
```

```
id :: forall a. a -> a
id = \ (@ a) (x :: a) -> x

idInt :: GHC.Types.Int -> GHC.Types.Int
idInt = id @ GHC.Types.Int
```

```
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g x = f (g x)
```

```
compose :: forall b c a. (b -> c) -> (a -> b) -> a -> c
compose = \ (@ b) (@ c) (@ a) (f1 :: b -> c) (g :: a -> b) (x1 :: a) -> f1 (g x1)
```

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

```
map :: forall a b. (a -> b) -> [a] -> [b]
map =
  \ (@ a) (@ b) (f :: a -> b) (xs :: [a]) ->
    case xs of _ {
      []      -> [] @ b;
      : y ys -> : @ b (f y) (map @ a @ b f ys)
    }
```

Machine generated names are created for a lot of transformation of Core. Generally they consist of a prefix and unique identifier. The prefix is often pass specific (i.e. ⓓˢ for desugar generated name s) and sometimes specific names are generated for specific automatically generated code. A non exhaustive cheat sheet is given below for deciphering what a name is and what it might stand for:

Prefix	Description
$f...	Dict-fun identifiers (from inst decls)
$dmop	Default method for 'op'
$wf	Worker for function 'f'
$sf	Specialised version of f
$gdm	Generated class method
$d	Dictionary names
$s	Specialized function name
$f	Foreign export
$pnC	n'th superclass selector for class C
T:C	Tycon for dictionary for class C
D:C	Data constructor for dictionary for class C
NTCo:T	Coercion for newtype T to its underlying runtime representation

Of important note is that the Λ and λ for type-level and value-level lambda abstraction are represented by the same symbol (⟍) in core, which is a simplifying detail of the GHC's implementation but a source of some confusion when starting.

```
-- System-F Notation
Λ b c a. λ (f1 : b -> c) (g : a -> b) (x1 : a). f1 (g x1)
```

```
-- Haskell Core
```

```
\ (@ b) (@ c) (@ a) (f1 :: b -> c) (g :: a -> b) (x1 :: a) -> f1 (g x1)
```

The `seq` function has an intuitive implementation in the Core language.

```
x `seq` y
```

```
case x of _ {
  __DEFAULT -> y
}
```

One particularly notable case of the Core desugaring process is that pattern matching on overloaded numbers implicitly translates into equality test (i.e. `Eq`).

```
f 0 = 1
f 1 = 2
f 2 = 3
f 3 = 4
f 4 = 5
f _ = 0


f :: forall a b. (Eq a, Num a, Num b) => a -> b
f =
  \ (@ a)
    (@ b)
    ($dEq :: Eq a)
    ($dNum :: Num a)
    ($dNum1 :: Num b)
    (ds :: a) ->
    case == $dEq ds (fromInteger $dNum (__integer 0)) of _ {
      False ->
        case == $dEq ds (fromInteger $dNum (__integer 1)) of _ {
          False ->
            case == $dEq ds (fromInteger $dNum (__integer 2)) of _ {
              False ->
                case == $dEq ds (fromInteger $dNum (__integer 3)) of _ {
                  False ->
                    case == $dEq ds (fromInteger $dNum (__integer 4)) of _ {
                      False -> fromInteger $dNum1 (__integer 0);
                      True -> fromInteger $dNum1 (__integer 5)
                    };
                  True -> fromInteger $dNum1 (__integer 4)
                };
              True -> fromInteger $dNum1 (__integer 3)
            };
          True -> fromInteger $dNum1 (__integer 2)
        };
      True -> fromInteger $dNum1 (__integer 1)
```

```
      }
```

Of course, adding a concrete type signature changes the desugar just matching on the unboxed values.

```
f :: Int -> Int
f =
  \ (ds :: Int) ->
    case ds of _ { I# ds1 ->
    case ds1 of _ {
      __DEFAULT -> I# 0;
      0 -> I# 1;
      1 -> I# 2;
      2 -> I# 3;
      3 -> I# 4;
      4 -> I# 5
    }
    }
```

See:

- Core Spec
- Core By Example
- CoreSynType

Inliner

```
infixr 0  $

($):: (a -> b) -> a -> b
f $ x =  f x
```

Having to enter a secondary closure every time we used `($)` would introduce an enormous overhead. Fortunately GHC has a pass to eliminate small functions like this by simply replacing the function call with the body of its definition at appropriate call-sites. There compiler contains a variety heuristics for determining when this kind of substitution is appropriate and the potential costs involved.

In addition to the automatic inliner, manual pragmas are provided for more granular control over inlining. It's important to note that naive inlining quite often results in significantly worse performance and longer compilation times.

```
{-# INLINE func #-}
{-# INLINABLE func #-}
{-# NOINLINE func #-}
```

For example the contrived case where we apply a binary function to two arguments. The function body is small and instead of entering another closure just to apply the given function, we could in fact just inline the function application at the call site.

```
{-# INLINE foo #-}
{-# NOINLINE bar #-}

foo :: (a -> b -> c) -> a -> b -> c
foo f x y = f x y

bar :: (a -> b -> c) -> a -> b -> c
bar f x y = f x y

test1 :: Int
test1 = foo (+) 10 20

test2 :: Int
test2 = bar (+) 20 30
```

Looking at the core, we can see that in `test2` the function has indeed been expanded at the call site and simply performs the addition there instead of another indirection.

```
test1 :: Int
test1 =
  let {
    f :: Int -> Int -> Int
    f = + $fNumInt } in
  let {
    x :: Int
    x = I# 10 } in
  let {
    y :: Int
    y = I# 20 } in
  f x y

test2 :: Int
test2 = bar (+ $fNumInt) (I# 20) (I# 30)
```

Cases marked with `NOINLINE` generally indicate that the logic in the function is using something like `unsafePerformIO` or some other unholy function. In these cases naive inlining might duplicate effects at multiple call-sites throughout the program which would be undesirable.

See:

- Secrets of the Glasgow Haskell Compiler inliner

Dictionaries

The Haskell language defines the notion of Typeclasses but is agnostic to how they are implemented in a Haskell compiler. GHC's particular implementation uses a pass called the *dictionary passing translation* part of the elaboration phase of the typechecker which translates Core functions with typeclass constraints into implicit parameters of which record-like structures containing the function implementations are passed.

```
class Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
```

This class can be thought as the implementation equivalent to the following parameterized record of functions.

```
data DNum a = DNum (a -> a -> a) (a -> a -> a) (a -> a)

add (DNum a m n) = a
mul (DNum a m n) = m
neg (DNum a m n) = n

numDInt :: DNum Int
numDInt = DNum plusInt timesInt negateInt

numDFloat :: DNum Float
numDFloat = DNum plusFloat timesFloat negateFloat
```

```
+ :: forall a. Num a => a -> a -> a
+ = \ (@ a) (tpl :: Num a) ->
  case tpl of _ { D:Num tpl _ _ -> tpl }

* :: forall a. Num a => a -> a -> a
* = \ (@ a) (tpl :: Num a) ->
  case tpl of _ { D:Num _ tpl _ -> tpl }

negate :: forall a. Num a => a -> a
negate = \ (@ a) (tpl :: Num a) ->
  case tpl of _ { D:Num _ _ tpl -> tpl }
```

Num and Ord have simple translation but for monads with existential type variables in their signatures, the only way to represent the equivalent dictionary is using RankNTypes. In addition a typeclass may also include superclasses which would be included in the typeclass dictionary and parameterized over the same arguments and an implicit superclass constructor function is created to pull out functions from the superclass for the current monad.

```
data DMonad m = DMonad
  { bind   :: forall a b. m a -> (a -> m b) -> m b
  , return :: forall a. a -> m a
  }
```

```
class (Functor t, Foldable t) => Traversable t where
    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```

```
    traverse f = sequenceA . fmap f
```

```
data DTraversable t = DTraversable
  { dFunctorTraversable :: DFunctor t  -- superclass dictionary
  , dFoldableTraversable :: DFoldable t -- superclass dictionary
  , traverse :: forall a. Applicative f => (a -> f b) -> t a -> f (t b)
  }
```

Indeed this is not that far from how GHC actually implements typeclasses. It elaborates into projection functions and data constructors nearly identical to this, and are implicitly threaded for every overloaded identifier.

Specialization

Overloading in Haskell is normally not entirely free by default, although with an optimization called specialization it can be made to have zero cost at specific points in the code where performance is crucial. This is not enabled by default by virtue of the fact that GHC is not a whole-program optimizing compiler and most optimizations (not all) stop at module boundaries.

GHC's method of implementing typeclasses means that explicit dictionaries are threaded around implicitly throughout the call sites. This is normally the most natural way to implement this functionality since it preserves separate compilation. A function can be compiled independently of where it is declared, not recompiled at every point in the program where it's called. The dictionary passing allows the caller to thread the implementation logic for the types to the call-site where it can then be used throughout the body of the function.

Of course this means that in order to get at a specific typeclass function we need to project (possibly multiple times) into the dictionary structure to pluck out the function reference. The runtime makes this very cheap but not entirely free.

Many C++ compilers or whole program optimizing compilers do the opposite however, they explicitly specialize each and every function at the call site replacing the overloaded function with its type-specific implementation. We can selectively enable this kind of behavior using class specialization.

```
module Specialize (spec, nonspec, f) where

{-# SPECIALIZE INLINE f :: Double -> Double -> Double #-}

f :: Floating a => a -> a -> a
f x y = exp (x + y) * exp (x + y)

nonspec :: Float
nonspec = f (10 :: Float) (20 :: Float)

spec :: Double
spec = f (10 :: Double) (20 :: Double)
```

Non-specialized

```
f :: forall a. Floating a => a -> a -> a
f =
```

```
\ (@ a) ($dFloating :: Floating a) (eta :: a) (eta1 :: a) ->
  let {
    a :: Fractional a
    a = $p1Floating @ a $dFloating } in
  let {
    $dNum :: Num a
    $dNum = $p1Fractional @ a a } in
  * @ a
    $dNum
    (exp @ a $dFloating (+ @ a $dNum eta eta1))
    (exp @ a $dFloating (+ @ a $dNum eta eta1))
```

In the specialized version the typeclass operations placed directly at the call site and are simply unboxed arithmetic. This will map to a tight set of sequential CPU instructions and is very likely the same code generated by C.

```
spec :: Double
spec = D# (*## (expDouble# 30.0) (expDouble# 30.0))
```

The non-specialized version has to project into the typeclass dictionary (`$fFloatingFloat`) 6 times and likely go through around 25 branches to perform the same operation.

```
nonspec :: Float
nonspec =
  f @ Float $fFloatingFloat (F# (__float 10.0)) (F# (__float 20.0))
```

For a tight loop over numeric types specializing at the call site can result in orders of magnitude performance increase. Although the cost in compile-time can often be non-trivial and when used function used at many call-sites this can slow GHC's simplifier pass to a crawl.

The best advice is profile and look for large uses of dictionary projection in tight loops and then specialize and inline in these places.

Using the `SPECIALISE INLINE` pragma can unintentionally cause GHC to diverge if applied over a recursive function, it will try to specialize itself infinitely.

Static Compilation

On Linux, Haskell programs can be compiled into a standalone statically linked binary that includes the runtime statically linked into it.

```
$ ghc -O2 --make -static -optc-static -optl-static -optl-pthread Example.hs
$ file Example
Example: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked,
$ ldd Example
        not a dynamic executable
```

In addition the file size of the resulting binary can be reduced by stripping unneeded symbols.

```
$ strip Example
```

Unboxed Types

The usual numerics types in Haskell can be considered to be a regular algebraic datatype with special constructor arguments for their underlying unboxed values. Normally unboxed types and explicit unboxing are not used in normal code, they are an implementation detail of the compiler and many optimizations exist to do the unboxing in a way that is guaranteed to be safe and preserve the high level semantics of Haskell. Nevertheless it is somewhat enlightening to understand how the types are implemented.

```
data Int = I# Int#

data Integer
  = S# Int#              -- Small integers
  | J# Int# ByteArray#   -- Large GMP integers

data Float = F# Float#
```

```
λ: :set -XMagicHash
λ: :m +GHC.Types
λ: :m +GHC.Prim

λ: :type 3#
3# :: GHC.Prim.Int#

λ: :type 3##
3## :: GHC.Prim.Word#

λ: :type 3.14#
3.14# :: GHC.Prim.Float#

λ: :type 3.14##
3.14## :: GHC.Prim.Double#

λ: :type 'c'#
'c'# :: GHC.Prim.Char#

λ: :type "Haskell"#
"Haskell"# :: Addr#

λ: :i Int
data Int = I# Int#       -- Defined in GHC.Types

λ: :k Int#
Int# :: #
```

An unboxed type with kind $#$ and will never unify a type variable of kind $*$. Intuitively a type with kind $*$ indicates a type with a uniform runtime representation that can be used polymorphically.

- *Lifted* - Can contain a bottom term, represented by a pointer. (`Int` , `Any` , `(,)`)
- *Unlited* - Cannot contain a bottom term, represented by a value on the stack. (`Int#` , `(#, #)`)

```
{-# LANGUAGE BangPatterns, MagicHash, UnboxedTuples #-}

import GHC.Exts
import GHC.Prim

ex1 :: Bool
ex1 = gtChar# a# b#
  where
    !(C# a#) = 'a'
    !(C# b#) = 'b'

ex2 :: Int
ex2 = I# (a# +# b#)
  where
    !(I# a#) = 1
    !(I# b#) = 2

ex3 :: Int
ex3 = (I# (1# +# 2# *# 3# +# 4#))


ex4 :: (Int, Int)
ex4 = (I# (dataToTag# False), I# (dataToTag# True))
```

The function for integer arithmetic used in the `Num` typeclass for `Int` is just pattern matching on this type to reveal the underlying unboxed value, performing the builtin arithmetic and then performing the packing up into `Int` again.

```
plusInt :: Int -> Int -> Int
(I# x) `plusInt` (I# y) = I# (x +# y)
```

Where `(+#)` is a low level function built into GHC that maps to intrinsic integer addition instruction for the CPU.

```
plusInt :: Int -> Int -> Int
plusInt a b = case a of {
    (I# a_) -> case b of {
      (I# b_) -> I# (+# a_ b_);
    };
};
```

Runtime values in Haskell are by default represented uniformly by a boxed `StgClosure*` struct which itself contains several payload values, which can themselves either be pointers to other boxed values or to unboxed literal values that fit within the system word size and are stored directly within the closure

In memory. The layout of the box is described by a bitmap in the header for the closure which describes which values in the payload are either pointers or non-pointers.

The `unpackClosure#` primop can be used to extract this information at runtime by reading off the bitmap on the closure.

```haskell
{-# LANGUAGE MagicHash, UnboxedTuples #-}
{-# OPTIONS_GHC -O1 #-}

module Main where

import GHC.Exts
import GHC.Base
import Foreign

data Size = Size
  { ptrs  :: Int
  , nptrs :: Int
  , size  :: Int
  } deriving (Show)

unsafeSizeof :: a -> Size
unsafeSizeof a =
  case unpackClosure# a of
    (# x, ptrs, nptrs #) ->
      let header  = sizeOf (undefined :: Int)
          ptr_c   = I# (sizeofArray# ptrs)
          nptr_c  = I# (sizeofByteArray# nptrs) `div` sizeOf (undefined :: Word)
          payload = I# (sizeofArray# ptrs +# sizeofByteArray# nptrs)
          size    = header + payload
      in Size ptr_c nptr_c size

data A = A {-# UNPACK #-} !Int
data B = B Int

main :: IO ()
main = do
  print (unsafeSizeof (A 42))
  print (unsafeSizeof (B 42))
```

For example the datatype with the `UNPACK` pragma contains 1 non-pointer and 0 pointers.

```haskell
data A = A {-# UNPACK #-} !Int
Size {ptrs = 0, nptrs = 1, size = 16}
```

While the default packed datatype contains 1 pointer and 0 non-pointers.

```haskell
data B = B Int
Size {ptrs = 1, nptrs = 0, size = 9}
```

The closure representation for data constructors are also "tagged" at the runtime with the tag of the specific constructor. This is however not a runtime type tag since there is no way to recover the type from the tag as all constructor simply use the sequence (0, 1, 2, ...). The tag is used to discriminate cases in pattern matching. The builtin `dataToTag#` can be used to pluck off the tag for an arbitrary datatype. This is used in some cases when desugaring pattern matches.

```
dataToTag# :: a -> Int#
```

For example:

```
-- data Bool = False | True
-- False ~ 0
-- True  ~ 1

a :: (Int, Int)
a = (I# (dataToTag# False), I# (dataToTag# True))
-- (0, 1)

-- data Ordering = LT | EQ | GT
-- LT ~ 0
-- EQ ~ 1
-- GT ~ 2

b :: (Int, Int, Int)
b = (I# (dataToTag# LT), I# (dataToTag# EQ), I# (dataToTag# GT))
-- (0, 1, 2)

-- data Either a b = Left a | Right b
-- Left ~ 0
-- Right ~ 1

c :: (Int, Int)
c = (I# (dataToTag# (Left 0)), I# (dataToTag# (Right 1)))
-- (0, 1)
```

String literals included in the source code are also translated into several primop operations. The `Addr#` type in Haskell stands for a static contagious buffer pre-allocated on the Haskell heap that can hold a `char*` sequence. The operation `unpackCString#` can scan this buffer and fold it up into a list of Chars from inside Haskell.

```
unpackCString# :: Addr# -> [Char]
```

This is done in the early frontend desugarer phase, where literals are translated into `Addr#` inline instead of giant chain of Cons'd characters. So our "Hello World" translates into the following Core:

```
-- print "Hello World"
```

```
print (unpackCString# "Hello World #)
```

See:

- [Unboxed Values as First-Class Citizens](#)

IO/ST

Both the IO and the ST monad have special state in the GHC runtime and share a very similar implementation. Both `ST a` and `IO a` are passing around an unboxed tuple of the form:

```
(# token, a #)
```

The `RealWorld#` token is "deeply magical" and doesn't actually expand into any code when compiled, but simply threaded around through every bind of the IO or ST monad and has several properties of being unique and not being able to be duplicated to ensure sequential IO actions are actually sequential. `unsafePerformIO` can thought of as the unique operation which discards the world token and plucks the `a` out, and is as the name implies not normally safe.

The `PrimMonad` abstracts over both these monads with an associated data family for the world token or ST thread, and can be used to write operations that generic over both ST and IO. This is used extensively inside of the vector package to allow vector algorithms to be written generically either inside of IO or ST.

```
{-# LANGUAGE MagicHash #-}
{-# LANGUAGE UnboxedTuples #-}

import GHC.IO ( IO(..) )
import GHC.Prim ( State#, RealWorld )
import GHC.Base ( realWorld# )

instance  Monad IO  where
    m >> k     = m >>= \ _ -> k
    return    = returnIO
    (>>=)     = bindIO
    fail s    = failIO s

returnIO :: a -> IO a
returnIO x = IO $ \ s -> (# s, x #)

bindIO :: IO a -> (a -> IO b) -> IO b
bindIO (IO m) k = IO $ \ s -> case m s of (# new_s, a #) -> unIO (k a) new_s

thenIO :: IO a -> IO b -> IO b
thenIO (IO m) k = IO $ \ s -> case m s of (# new_s, _ #) -> unIO k new_s

unIO :: IO a -> (State# RealWorld -> (# State# RealWorld, a #))
unIO (IO a) = a
```

```
{-# LANGUAGE MagicHash #-}
```

```haskell
{-# LANGUAGE UnboxedTuples #-}
{-# LANGUAGE TypeFamilies #-}

import GHC.IO ( IO(..) )
import GHC.ST ( ST(..) )
import GHC.Prim ( State#, RealWorld )
import GHC.Base ( realWorld# )

class Monad m => PrimMonad m where
  type PrimState m
  primitive :: (State# (PrimState m) -> (# State# (PrimState m), a #)) -> m a
  internal :: m a -> State# (PrimState m) -> (# State# (PrimState m), a #)

instance PrimMonad IO where
  type PrimState IO = RealWorld
  primitive = IO
  internal (IO p) = p

instance PrimMonad (ST s) where
  type PrimState (ST s) = s
  primitive = ST
  internal (ST p) = p
```

See:

- [Evaluation order and state tokens](#)

ghc-heap-view

Through some dark runtime magic we can actually inspect the `StgClosure` structures at runtime using various C and Cmm hacks to probe at the fields of the structure's representation to the runtime. The library `ghc-heap-view` can be used to introspect such things, although there is really no use for this kind of thing in everyday code it is very helpful when studying the GHC internals to be able to inspect the runtime implementation details and get at the raw bits underlying all Haskell types.

```haskell
{-# LANGUAGE MagicHash #-}

import GHC.Exts
import GHC.HeapView

import System.Mem

main :: IO ()
main = do
  -- Constr
  clo <- getClosureData $! ([1,2,3] :: [Int])
  print clo

  -- Thunk
  let thunk = id (1+1)
  clo <- getClosureData thunk
  print clo
```

```
   -- evaluate to WHNF
   thunk `seq` return ()

   -- Indirection
   clo <- getClosureData thunk
   print clo

   -- force garbage collection
   performGC

   -- Value
   clo <- getClosureData thunk
   print clo
```

A constructor (in this for cons constructor of list type) is represented by a ⟨CONSTR⟩ closure that holds two pointers to the head and the tail. The integer in the head argument is a static reference to the pre-allocated number and we see a single static reference in the SRT (static reference table).

```
ConsClosure {
  info = StgInfoTable {
    ptrs = 2,
    nptrs = 0,
    tipe = CONSTR_2_0,
    srtlen = 1
  },
  ptrArgs = [0x000000000074aba8/1,0x00007fca10504260/2],
  dataArgs = [],
  pkg = "ghc-prim",
  modl = "GHC.Types",
  name = ":"
}
```

We can also observe the evaluation and update of a thunk in process (⟨id (1+1)⟩). The initial thunk is simply a thunk type with a pointer to the code to evaluate it to a value.

```
ThunkClosure {
  info = StgInfoTable {
    ptrs = 0,
    nptrs = 0,
    tipe = THUNK,
    srtlen = 9
  },
  ptrArgs = [],
  dataArgs = []
}
```

When forced it is then evaluated and replaced with an Indirection closure which points at the computed value.

```
BlackholeClosure {
  info = StgInfoTable {
    ptrs = 1,
    nptrs = 0,
    tipe = BLACKHOLE,
    srtlen = 0
  },
  indirectee = 0x00007fca10511e88/1
}
```

When the copying garbage collector passes over the indirection, it then simply replaces the indirection with a reference to the actual computed value computed by `indirectee` so that future access does need to chase a pointer through the indirection pointer to get the result.

```
ConsClosure {
  info = StgInfoTable {
    ptrs = 0,
    nptrs = 1,
    tipe = CONSTR_0_1,
    srtlen = 0
  },
  ptrArgs = [],
  dataArgs = [2],
  pkg = "integer-gmp",
  modl = "GHC.Integer.Type",
  name = "S#"
}
```

STG

After being compiled into Core, a program is translated into a very similar intermediate form known as STG (Spineless Tagless G-Machine) an abstract machine model that makes all laziness explicit. The spineless indicates that function applications in the language do not have a spine of applications of functions are collapsed into a sequence of arguments. Currying is still present in the semantics since arity information is stored and partially applied functions will evaluate differently than saturated functions.

```
-- Spine
f x y z = App (App (App f x) y) z

-- Spineless
f x y z = App f [x, y, z]
```

All let statements in STG bind a name to a *lambda form*. A lambda form with no arguments is a thunk, while a lambda-form with arguments indicates that a closure is to be allocated that captures the variables explicitly mentioned.

Thunks themselves are either reentrant (`\r`) or updatable (`\u`) indicating that the thunk and either yields a value to the stack or is allocated on the heap after the update frame is evaluated All subsequent entry's of the thunk will yield the already-computed value without needing to redo the same

WVII.

A lambda form also indicates the *static reference table* a collection of references to static heap allocated values referred to by the body of the function.

For example turning on `-ddump-stg` we can see the expansion of the following compose function.

```
-- Frontend
compose f g = \x -> f (g x)
```

```
-- Core
compose :: forall t t1 t2. (t1 -> t) -> (t2 -> t1) -> t2 -> t
compose =
  \ (@ t) (@ t1) (@ t2) (f :: t1 -> t) (g :: t2 -> t1) (x :: t2) ->
    f (g x)
```

```
-- STG
compose :: forall t t1 t2. (t1 -> t) -> (t2 -> t1) -> t2 -> t =
    \r [f g x] let { sat :: t1 = \u [] g x; } in  f sat;
SRT(compose): []
```

For a more sophisticated example, let's trace the compilation of the factorial function.

```
-- Frontend
fac :: Int -> Int -> Int
fac a 0 = a
fac a n = fac (n*a) (n-1)
```

```
-- Core
Rec {
fac :: Int -> Int -> Int
fac =
  \ (a :: Int) (ds :: Int) ->
    case ds of wild { I# ds1 ->
    case ds1 of _ {
      __DEFAULT ->
        fac (* @ Int $fNumInt wild a) (- @ Int $fNumInt wild (I# 1));
      0 -> a
    }
    }
end Rec }
```

```
-- STG
fac :: Int -> Int -> Int =
```

```
    \r srt:(0,*bitmap*) [a ds]
        case ds of wild {
          I# ds1 ->
              case ds1 of _ {
                __DEFAULT ->
                    let {
                      sat :: Int =
                          \u srt:(1,*bitmap*) []
                              let { sat :: Int = NO_CCS I#! [1]; } in  - $fNumInt wild
                      let { sat :: Int = \u srt:(1,*bitmap*) [] * $fNumInt wild a;
                      } in  fac sat sat;
                    0 -> a;
                };
        };
 SRT(fac): [fac, $fNumInt]
```

Notice that the factorial function allocates two thunks (look for $\backslash u$) inside of the loop which are updated when computed. It also includes static references to both itself (for recursion) and the dictionary for instance of Num typeclass over the type Int .

Worker/Wrapper

With −O2 turned on GHC will perform a special optimization known as the Worker-Wrapper transformation which will split the logic of the factorial function across two definitions, the worker will operate over stack unboxed allocated machine integers which compiles into a tight inner loop while the wrapper calls into the worker and collects the end result of the loop and packages it back up into a boxed heap value. This can often be an order of of magnitude faster than the naive implementation which needs to pack and unpack the boxed integers on every iteration.

```
-- Worker
$wfac :: Int# -> Int# -> Int# =
    \r [ww ww1]
        case ww1 of ds {
          __DEFAULT ->
              case -# [ds 1] of sat {
                __DEFAULT ->
                    case *# [ds ww] of sat { __DEFAULT -> $wfac sat sat; };
              };
          0 -> ww;
        };
SRT($wfac): []

-- Wrapper
fac :: Int -> Int -> Int =
    \r [w w1]
        case w of _ {
          I# ww ->
              case w1 of _ {
                I# ww1 -> case $wfac ww ww1 of ww2 { __DEFAULT -> I# [ww2]; };
              };
        };
SRT(fac): []
```

See:

- [Writing Haskell as Fast as C](#)

Z-Encoding

The Z-encoding is Haskell's convention for generating names that are safely represented in the compiler target language. Simply put the z-encoding renames many symbolic characters into special sequences of the z character.

String	Z-Encoded String
`foo`	`foo`
`z`	`zz`
`Z`	`ZZ`
`.`	`.`
`()`	`Z0T`
`(,)`	`Z2T`
`(,,)`	`Z3T`
`_`	`zu`
`(`	`ZL`
`)`	`ZR`
`:`	`ZC`
`#`	`zh`
`.`	`zi`
`(#,#)`	`Z2H`
`(->)`	`ZLzmzgZR`

In this way we don't have to generate unique unidentifiable names for character rich names and can simply have a straightforward way to translate them into something unique but identifiable.

So for some example names from GHC generated code:

Z-Encoded String	Decoded String
`ZCMain_main_closure`	`:Main_main_closure`
`base_GHCziBase_map_closure`	`base_GHC.Base_map_closure`
`base_GHCziInt_I32zh_con_info`	`base_GHC.Int_I32#_con_info`
`ghczmprim_GHCziTuple_Z3T_con_info`	`ghc-prim_GHC.Tuple_(,,)_con_in`
`ghczmprim_GHCziTypes_ZC_con_info`	`ghc-prim_GHC.Types_:_con_info`

Cmm

Cmm is GHC's complex internal intermediate representation that maps directly onto the generated code for the compiler target. Cmm code code generated from Haskell is CPS-converted, all functions never return a value, they simply call the next frame in the continuation stack. All evaluation of functions proceed by indirectly jumping to a code object with its arguments placed on the stack by the caller.

This is drastically different than C's evaluation model, where are placed on the stack and a function yields a value to the stack after it returns.

There are several common suffixes you'll see used in all closures and function names:

Symbol	Meaning
`0`	No argument
`p`	Garage Collected Pointer

n	Word-sized non-pointer
l	64-bit non-pointer (long)
v	Void
f	Float
d	Double
v16	16-byte vector
v32	32-byte vector
v64	64-byte vector

Cmm Registers

There are 10 registers that described in the machine model. **Sp** is the pointer to top of the stack, **SpLim** is the pointer to last element in the stack. **Hp** is the heap pointer, used for allocation and garbage collection with **HpLim** the current heap limit.

The **R1** register always holds the active closure, and subsequent registers are arguments passed in registers. Functions with more than 10 values spill into memory.

- Sp
- SpLim
- Hp
- HpLim
- HpAlloc
- R1
- R2
- R3
- R4
- R5
- R6
- R7
- R8
- R9
- R10

Examples

To understand Cmm it is useful to look at the code generated by the equivalent Haskell and slowly understand the equivalence and mechanical translation maps one to the other.

There are generally two parts to every Cmm definition, the **info table** and the **entry code**. The info table maps directly `StgInfoTable` struct and contains various fields related to the type of the closure, its payload, and references. The code objects are basic blocks of generated code that correspond to the logic of the Haskell function/constructor.

For the simplest example consider a constant static constructor. Simply a function which yields the Unit value. In this case the function is simply a constructor with no payload, and is statically allocated.

Haskell:

```
unit = ()
```

Cmm:

```
[section "data" {
     unit_closure:
          const ()_static_info;
  }]
```

Consider a static constructor with an argument.

Haskell:

```haskell
con :: Maybe ()
con = Just ()
```

Cmm:

```
[section "data" {
    con_closure:
        const Just_static_info;
        const ()_closure+1;
        const 1;
  }]
```

Consider a literal constant. This is a static value.

Haskell:

```haskell
lit :: Int
lit = 1
```

Cmm:

```
[section "data" {
    lit_closure:
        const I#_static_info;
        const 1;
  }]
```

Consider the identity function.

Haskell:

```haskell
id x = x
```

Cmm:

```
[section "data" {
    id_closure:
```

```
              const id_info;
    },
    id_info()
            { label: id_info
              rep:HeapRep static { Fun {arity: 1 fun_type: ArgSpec 5} }
            }
        ch1:
            R1 = R2;
            jump stg_ap_0_fast; // [R1]
    }]
```

Consider the constant function.

Haskell:

```
constant x y = x
```

Cmm:

```
[section "data" {
      constant_closure:
            const constant_info;
    },
    constant_info()
            { label: constant_info
              rep:HeapRep static { Fun {arity: 2 fun_type: ArgSpec 12} }
            }
        cgT:
            R1 = R2;
            jump stg_ap_0_fast; // [R1]
    }]
```

Consider a function where application of a function (of unknown arity) occurs.

Haskell:

```
compose f g x = f (g x)
```

Cmm:

```
[section "data" {
      compose_closure:
            const compose_info;
    },
    compose_info()
            { label: compose_info
```

```
      rep:HeapRep static { Fun {arity: 3 fun_type: ArgSpec 20} }
        }
    ch9:
        Hp = Hp + 32;
        if (Hp > HpLim) goto chd;
        I64[Hp - 24] = stg_ap_2_upd_info;
        I64[Hp - 8] = R3;
        I64[Hp + 0] = R4;
        R1 = R2;
        R2 = Hp - 24;
        jump stg_ap_p_fast; // [R1, R2]
    che:
        R1 = compose_closure;
        jump stg_gc_fun; // [R1, R4, R3, R2]
    chd:
        HpAlloc = 32;
        goto che;
  }]
```

Consider a function which branches using pattern matching:

Haskell:

```
match :: Either a a -> a
match x = case x of
  Left a -> a
  Right b -> b
```

Cmm:

```
[section "data" {
    match_closure:
        const match_info;
 },
 sio_ret()
        { label: sio_info
          rep:StackRep []
        }
    ciL:
        _ciM::I64 = R1 & 7;
        if (_ciM::I64 >= 2) goto ciN;
        R1 = I64[R1 + 7];
        Sp = Sp + 8;
        jump stg_ap_0_fast; // [R1]
    ciN:
        R1 = I64[R1 + 6];
        Sp = Sp + 8;
        jump stg_ap_0_fast; // [R1]
 },
 match_info()
```

```
        { label: match_info
            rep:HeapRep static { Fun {arity: 1 fun_type: ArgSpec 5} }
        }
    ciP:
        if (Sp - 8 < SpLim) goto ciR;
        R1 = R2;
        I64[Sp - 8] = sio_info;
        Sp = Sp - 8;
        if (R1 & 7 != 0) goto ciU;
        jump I64[R1]; // [R1]
    ciR:
        R1 = match_closure;
        jump stg_gc_fun; // [R1, R2]
    ciU: jump sio_info; // [R1]
}]
```

Macros

Cmm itself uses many macros to stand for various constructs, many of which are defined in an external C header file. A short reference for the common types:

Cmm	Description
C_	char
D_	double
F_	float
W_	word
P_	garbage collected pointer
I_	int
L_	long
FN_	function pointer (no arguments)
EF_	extern function pointer
I8	8-bit integer
I16	16-bit integer
I32	32-bit integer
I64	64-bit integer

Many of the predefined closures (stg_ap_p_fast , etc) are themselves mechanically generated and more or less share the same form (a giant switch statement on closure type, update frame, stack adjustment). Inside of GHC is a file named GenApply.hs that generates most of these functions. See the Gist link in the reading section for the current source file that GHC generates. For example the output for stg_ap_p_fast .

```
stg_ap_p_fast
{   W_ info;
    W_ arity;
    if (GETTAG(R1)==1) {
        Sp_adj(0);
        jump %GET_ENTRY(R1-1) [R1,R2];
    }
    if (Sp - WDS(2) < SpLim) {
        Sp_adj(-2);
        W_[Sp+WDS(1)] = R2;
        Sp(0) = stg_ap_p_info;
```

```
            jump __stg_gc_enter_1 [R1];
        }
        R1 = UNTAG(R1);
        info = %GET_STD_INFO(R1);
        switch [INVALID_OBJECT .. N_CLOSURE_TYPES] (TO_W_(%INFO_TYPE(info))) {
            case FUN,
                 FUN_1_0,
                 FUN_0_1,
                 FUN_2_0,
                 FUN_1_1,
                 FUN_0_2,
                 FUN_STATIC: {
                arity = TO_W_(StgFunInfoExtra_arity(%GET_FUN_INFO(R1)));
                ASSERT(arity > 0);
                if (arity == 1) {
                    Sp_adj(0);
                    R1 = R1 + 1;
                    jump %GET_ENTRY(UNTAG(R1)) [R1,R2];
                } else {
                    Sp_adj(-2);
                    W_[Sp+WDS(1)] = R2;
                    if (arity < 8) {
                      R1 = R1 + arity;
                    }
                    BUILD_PAP(1,1,stg_ap_p_info,FUN);
                }
            }
            default: {
                Sp_adj(-2);
                W_[Sp+WDS(1)] = R2;
                jump RET_LBL(stg_ap_p) [];
            }
        }
    }
}
```

Handwritten Cmm can be included in a module manually by first compiling it through GHC into an object and then using a special FFI invocation.

```
#include "Cmm.h"

factorial {
  entry:
      W_ n  ;
      W_ acc;
      n = R1 ;
      acc = n ;
      n = n - 1 ;

  for:
      if (n <= 0 ) {
          RET_N(acc);
      } else {
```

```
            acc = acc * n  ;
            n = n - 1 ;
            goto for ;
        }
        RET_N(0);
}
```

```
-- ghc -c factorial.cmm -o factorial.o
-- ghc factorial.o Example.hs -o Example

{-# LANGUAGE MagicHash #-}
{-# LANGUAGE UnliftedFFITypes #-}
{-# LANGUAGE GHCForeignImportPrim #-}
{-# LANGUAGE ForeignFunctionInterface #-}

module Main where

import GHC.Prim
import GHC.Word

foreign import prim "factorial" factorial_cmm  :: Word# -> Word#

factorial :: Word64 -> Word64
factorial (W64# n) =  W64# (factorial_cmm n)

main :: IO ()
main = print (factorial 5)
```

See:

- CmmType
- MiscClosures
- StgCmmArgRep

Cmm Runtime:

- Apply.cmm
- StgStdThunks.cmm
- StgMiscClosures.cmm
- PrimOps.cmm
- Updates.cmm
- Precompiled Closures (Autogenerated Output)

Optimization Hacks

Tables Next to Code

GHC will place the info table for a toplevel closure directly next to the entry-code for the objects in memory such that the fields from the info table can be accessed by pointer arithmetic on the function pointer to the code itself. Not performing this optimization would involve chasing through one more pointer to get to the info table. Given how often info-tables are accessed using the tables-next-to-code optimization results in a tractable speedup.

Pointer Tagging

Depending on the type of the closure involved, GHC will utilize the last few bits in a pointer to the closure to store information that can be read off from the bits of pointer itself before jumping into or access the info tables. For thunks this can be information like whether it is evaluated to WHNF or not, for constructors it contains the constructor tag (if it fits) to avoid an info table lookup.

Depending on the architecture the tag bits are either the last 2 or 3 bits of a pointer.

```
// 32 bit arch
TAG_BITS = 2


// 64-bit arch
TAG_BITS = 3
```

These occur in Cmm most frequently via the following macro definitions:

```
#define TAG_MASK ((1 << TAG_BITS) - 1)
#define UNTAG(p) (p & ~TAG_MASK)
#define GETTAG(p) (p & TAG_MASK)
```

So for instance in many of the precompiled functions, there will be a test for whether the active closure $R1$ is already evaluated.

```
if (GETTAG(R1)==1) {
    Sp_adj(0);
    jump %GET_ENTRY(R1-1) [R1,R2];
}
```

Profiling

EKG

EKG is a monitoring tool that can monitor various aspect of GHC's runtime alongside an active process. The interface for the output is viewable within a browser interface. The monitoring process is forked off (in a system thread) from the main process.

```
{-# Language OverloadedStrings #-}

import Control.Monad
import System.Remote.Monitoring

main :: IO ()
main = do
  ekg <- forkServer "localhost" 8000
  putStrLn "Started server on http://localhost:8000"
  forever $ getLine >>= putStrLn
```

ekg

Allocation rate
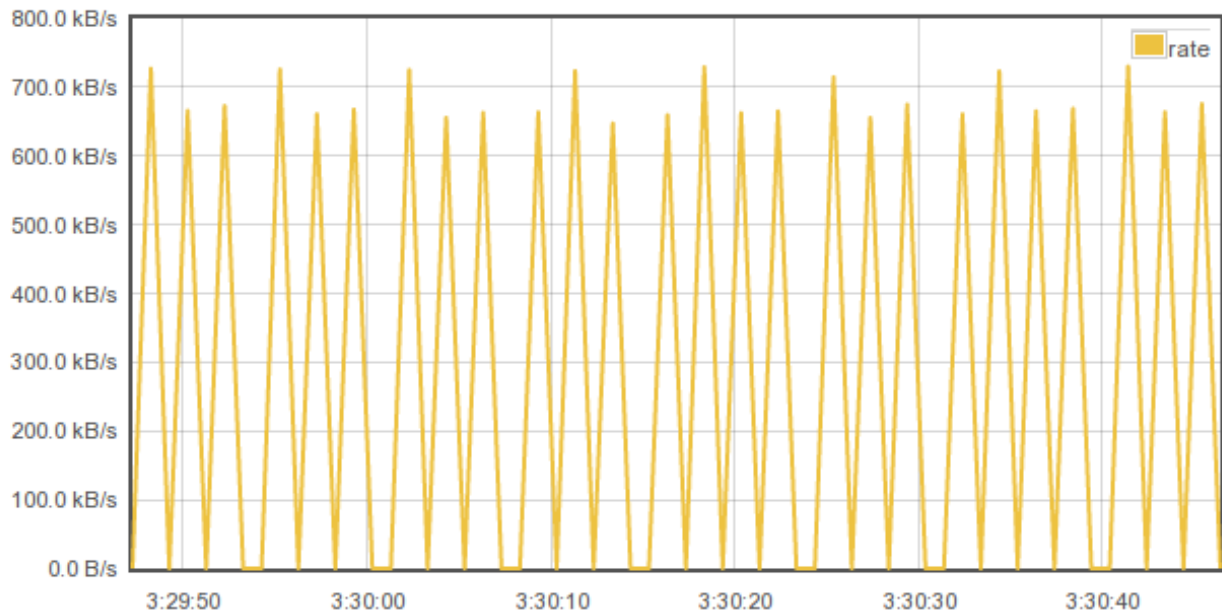
RTS Profiling

The GHC runtime system can be asked to dump information about

```
$ ./program +RTS -s

        1,939,784 bytes allocated in the heap
           11,160 bytes copied during GC
           44,416 bytes maximum residency (2 sample(s))
           21,120 bytes maximum slop
                1 MB total memory in use (0 MB lost due to fragmentation)

                                    Tot time (elapsed)  Avg pause  Max pause
  Gen  0         2 colls,     0 par    0.00s    0.00s     0.0000s    0.0000s
  Gen  1         2 colls,     0 par    0.00s    0.00s     0.0002s    0.0003s

  INIT    time    0.00s  (  0.00s elapsed)
  MUT     time    0.00s  (  0.01s elapsed)
  GC      time    0.00s  (  0.00s elapsed)
  EXIT    time    0.00s  (  0.00s elapsed)
  Total   time    0.01s  (  0.01s elapsed)

  %GC     time       5.0%  (7.1% elapsed)

  Alloc rate    398,112,898 bytes per MUT second

  Productivity  91.4% of total user, 128.8% of total elapsed
```

Productivity indicates the amount of time spent during execution compared to the time spent garbage collecting. Well tuned CPU bound programs are often in the 90-99% range of productivity range.

In addition individual function profiling information can be generated by compiling the program with `‑prof` flag. The resulting information is outputted to a `.prof` file of the same name as the module. This is useful for tracking down hotspots in the program.

```
$ ghc -O2 program.hs -prof -auto-all
$ ./program +RTS -p
$ cat program.prof
        Mon Oct 27 23:00 2014 Time and Allocation Profiling Report  (Final)

           program +RTS -p -RTS

        total time  =        0.01 secs   (7 ticks @ 1000 us, 1 processor)
        total alloc =   1,937,336 bytes  (excludes profiling overheads)

COST CENTRE MODULE              %time %alloc

CAF         Main              100.0   97.2
CAF         GHC.IO.Handle.FD    0.0    1.8


                                                 individual     inherited
COST CENTRE MODULE                  no.   entries  %time %alloc  %time %alloc

MAIN        MAIN                     42        0    0.0    0.7  100.0  100.0
 CAF        Main                     83        0  100.0   97.2  100.0   97.2
 CAF        GHC.IO.Encoding          78        0    0.0    0.1    0.0    0.1
 CAF        GHC.IO.Handle.FD         77        0    0.0    1.8    0.0    1.8
 CAF        GHC.Conc.Signal          74        0    0.0    0.0    0.0    0.0
 CAF        GHC.IO.Encoding.Iconv    69        0    0.0    0.0    0.0    0.0
 CAF        GHC.Show                 60        0    0.0    0.0    0.0    0.0
```

Languages

Unbound

Several libraries exist to mechanize the process of writing name capture and substitution, since it is largely mechanical. Probably the most robust is the `unbound` library. For example we can implement the infer function for a small Hindley-Milner system over a simple typed lambda calculus without having to write the name capture and substitution mechanics ourselves.

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE OverloadedStrings #-}

module Infer where

import Data.String
import Data.Map (Map)
import Control.Monad.Error
```

```haskell
import qualified Data.Map as Map

import qualified Unbound.LocallyNameless as NL
import Unbound.LocallyNameless hiding (Subst, compose)

data Type
  = TVar (Name Type)
  | TArr Type Type
  deriving (Show)

data Expr
  = Var (Name Expr)
  | Lam (Bind (Name Expr) Expr)
  | App Expr Expr
  | Let (Bind (Name Expr) Expr)
  deriving (Show)

$(derive [''Type, ''Expr])


instance IsString Expr where
    fromString = Var . fromString
instance IsString Type where
    fromString = TVar . fromString
instance IsString (Name Expr) where
    fromString = string2Name
instance IsString (Name Type) where
    fromString = string2Name

instance Eq Type where
    (==) = eqType

eqType :: Type -> Type -> Bool
eqType (TVar v1) (TVar v2) = v1 == v2
eqType _ _ = False

uvar :: String -> Expr
uvar x = Var (s2n x)

tvar :: String -> Type
tvar x = TVar (s2n x)

instance Alpha Type
instance Alpha Expr

instance NL.Subst Type Type where
  isvar (TVar v) = Just (SubstName v)
  isvar _ = Nothing

instance NL.Subst Expr Expr where
  isvar (Var v) = Just (SubstName v)
  isvar _ = Nothing


instance NL.Subst Expr Type where
```

```haskell
data TypeError
  = UnboundVariable (Name Expr)
  | GenericTypeError
  deriving (Show)

instance Error TypeError where
  noMsg = GenericTypeError


type Env = Map (Name Expr) Type
type Constraint = (Type, Type)
type Infer = ErrorT TypeError FreshM

empty :: Env
empty = Map.empty

freshtv :: Infer Type
freshtv = do
  x <- fresh "_t"
  return $ TVar x

infer :: Env -> Expr -> Infer (Type, [Constraint])
infer env expr = case expr of

  Lam b -> do
    (n,e) <- unbind b
    tv <- freshtv
    let env' = Map.insert n tv env
    (t, cs) <- infer env' e
    return (TArr tv t, cs)

  App e1 e2 -> do
    (t1, cs1) <- infer env e1
    (t2, cs2) <- infer env e2
    tv <- freshtv
    return (tv, (t1, TArr t2 tv) : cs1 ++ cs2)

  Var n -> do
    case Map.lookup n env of
      Nothing -> throwError $ UnboundVariable n
      Just t  -> return (t, [])

  Let b -> do
    (n, e) <- unbind b
    (tBody, csBody) <- infer env e
    let env' = Map.insert n tBody env
    (t, cs) <- infer env' e
    return (t, cs ++ csBody)
```

Unbound Generics

Recently unbound was ported to use GHC.Generics instead of Template Haskell. The API is effectively the same, so for example a simple lambda calculus could be written as:

```haskell
{-# LANGUAGE DeriveGeneric #-}
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE ScopedTypeVariables #-}

module LC where

import Unbound.Generics.LocallyNameless
import Unbound.Generics.LocallyNameless.Internal.Fold (toListOf)

import GHC.Generics

import Data.Typeable (Typeable)
import Data.Set as S

import Control.Monad.Reader (Reader, runReader)

data Exp
  = Var (Name Exp)
  | Lam (Bind (Name Exp) Exp)
  | App Exp Exp
  deriving (Show, Generic, Typeable)

instance Alpha Exp

instance Subst Exp Exp where
  isvar (Var x) = Just (SubstName x)
  isvar _       = Nothing

fvSet :: (Alpha a, Typeable b) => a -> S.Set (Name b)
fvSet = S.fromList . toListOf fv

type M a = FreshM a

(=~) :: Exp -> Exp -> M Bool
e1 =~ e2 | e1 `aeq` e2 = return True
e1 =~ e2 = do
    e1' <- red e1
    e2' <- red e2
    if e1' `aeq` e1 && e2' `aeq` e2
      then return False
      else e1' =~ e2'

-- Reduction
red :: Exp -> M Exp
red (App e1 e2) = do
  e1' <- red e1
  e2' <- red e2
```

```
    case e1 of
      Lam bnd -> do
           (x, e1'') <- unbind bnd
           return $ subst x e2' e1''
      otherwise -> return $ App e1' e2'
red (Lam bnd) = do
    (x, e) <- unbind bnd
    e' <- red e
    case e of
      App e1 (Var y) | y == x && x `S.notMember` fvSet e1 -> return e1
      otherwise -> return (Lam (bind x e'))
red (Var x) = return $ (Var x)


x :: Name Exp
x = string2Name "x"

y :: Name Exp
y = string2Name "y"

z :: Name Exp
z = string2Name "z"

s :: Name Exp
s = string2Name "s"

lam :: Name Exp -> Exp -> Exp
lam x y = Lam (bind x y)

zero  = lam s (lam z (Var z))
one   = lam s (lam z (App (Var s) (Var z)))
two   = lam s (lam z (App (Var s) (App (Var s) (Var z))))
three = lam s (lam z (App (Var s) (App (Var s) (App (Var s) (Var z))))))

plus = lam x (lam y (lam s (lam z (App (App (Var x) (Var s)) (App (App (Var y) (Var s)

true = lam x (lam y (Var x))
false = lam x (lam y (Var y))
if_ x y z = (App (App x y) z)

main :: IO ()
main = do
  print $ lam x (Var x) `aeq` lam y (Var y)
  print $ not (lam x (Var y) `aeq` lam x (Var x))
  print $ lam x (App (lam y (Var x)) (lam y (Var y))) =~ (lam y (Var y))
  print $ lam x (App (Var y) (Var x)) =~ Var y
  print $ if_ true (Var x) (Var y) =~ Var x
  print $ if_ false (Var x) (Var y) =~ Var y
  print $ App (App plus one) two =~ three
```

See:

LLVM

LLVM is a library for generating machine code. The llvm-general bindings provide a way to model, compile and execute LLVM bytecode from within the Haskell runtime.

See:

- Implementing a JIT Compiled Language with Haskell and LLVM

Printer Combinators

Pretty printer combinators compose logic to print strings.

	Combinators
`<>`	Concatenation
`<+>`	Spaced concatenation
`char`	Renders a character as a `Doc`
`text`	Renders a string as a `Doc`

```haskell
{-# LANGUAGE FlexibleInstances #-}

import Text.PrettyPrint
import Text.Show.Pretty (ppShow)

parensIf ::  Bool -> Doc -> Doc
parensIf True = parens
parensIf False = id

type Name = String

data Expr
  = Var String
  | Lit Ground
  | App Expr Expr
  | Lam Name Expr
  deriving (Eq, Show)

data Ground
  = LInt Int
  | LBool Bool
  deriving (Show, Eq, Ord)


class Pretty p where
  ppr :: Int -> p -> Doc

instance Pretty String where
  ppr _ x = text x

instance Pretty Expr where
  ppr _ (Var x)        = text x
  ppr _ (Lit (LInt a))  = text (show a)
  ppr _ (Lit (LBool b)) = text (show b)
```

```haskell
    ppr p e@(App _ _) =
      let (f, xs) = viewApp e in
      let args = sep $ map (ppr (p+1)) xs in
      parensIf (p>0) $ ppr p f <+> args

    ppr p e@(Lam _ _) =
      let body = ppr (p+1) (viewBody e) in
      let vars = map (ppr 0) (viewVars e) in
      parensIf (p>0) $ char '\\' <> hsep vars <+> text "." <+> body

viewVars :: Expr -> [Name]
viewVars (Lam n a) = n : viewVars a
viewVars _ = []

viewBody :: Expr -> Expr
viewBody (Lam _ a) = viewBody a
viewBody x = x

viewApp :: Expr -> (Expr, [Expr])
viewApp (App e1 e2) = go e1 [e2]
  where
    go (App a b) xs = go a (b : xs)
    go f xs = (f, xs)

ppexpr :: Expr -> String
ppexpr = render . ppr 0


s, k, example :: Expr
s = Lam "f" (Lam "g" (Lam "x" (App (Var "f") (App (Var "g") (Var "x")))))
k = Lam "x" (Lam "y" (Var "x"))
example = App s k

main :: IO ()
main = do
  putStrLn $ ppexpr s
  putStrLn $ ppShow example
```

The pretty printed form of the ⓚ combinator:

```
\f g x . (f (g x))
```

The `Text.Show.Pretty` library can be used to pretty print nested data structures in a more human readable form for any type that implements `Show`. For example a dump of the structure for the AST of SK combinator with `ppShow`.

```
App
  (Lam
     "f" (Lam "g" (Lam "x" (App (Var "f") (App (Var "g") (Var "x"))))))
```

```
    (Lam    x    (Lam    y    (Var    x  )))
```

Adding the following to your ghci.conf can be useful for working with deeply nested structures interactively.

```haskell
import Text.Show.Pretty (ppShow)
let pprint x = putStrLn $ ppShow x
```

See: The Design of a Pretty-printing Library

Haskeline

Haskeline is cross-platform readline support which plays nice with GHCi as well.

```haskell
runInputT :: Settings IO -> InputT IO a -> IO a
getInputLine :: String -> InputT IO (Maybe String)
```

```haskell
import Control.Monad.Trans
import System.Console.Haskeline

type Repl a = InputT IO a

process :: String -> IO ()
process = putStrLn

repl :: Repl ()
repl = do
  minput <- getInputLine "Repl> "
  case minput of
    Nothing -> outputStrLn "Goodbye."
    Just input -> (liftIO $ process input) >> repl

main :: IO ()
main = runInputT defaultSettings repl
```

Template Haskell

Quasiquotation

Quasiquotation allows us to express "quoted" blocks of syntax that need not necessarily be be the syntax of the host language, but unlike just writing a giant string it is instead parsed into some AST datatype in the host language. Notably values from the host languages can be injected into the custom language via user-definable logic allowing information to flow between the two languages.

In practice quasiquotation can be used to implement custom domain specific languages or integrate with other general languages entirely via code-generation.

We've already seen how to write a Parsec parser, now let's write a quasiquoter for it.

```haskell
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}

module Quasiquote where

import Language.Haskell.TH
import Language.Haskell.TH.Syntax
import Language.Haskell.TH.Quote

import Text.Parsec
import Text.Parsec.String (Parser)
import Text.Parsec.Language (emptyDef)

import qualified Text.Parsec.Expr as Ex
import qualified Text.Parsec.Token as Tok

import Control.Monad.Identity

data Expr
  = Tr
  | Fl
  | Zero
  | Succ Expr
  | Pred Expr
  deriving (Eq, Show)

instance Lift Expr where
  lift Tr        = [| Tr |]
  lift Fl        = [| Tr |]
  lift Zero      = [| Zero |]
  lift (Succ a)  = [| Succ a |]
  lift (Pred a)  = [| Pred a |]

type Op = Ex.Operator String () Identity

lexer :: Tok.TokenParser ()
lexer = Tok.makeTokenParser emptyDef

parens :: Parser a -> Parser a
parens = Tok.parens lexer

reserved :: String -> Parser ()
reserved = Tok.reserved lexer

semiSep :: Parser a -> Parser [a]
semiSep = Tok.semiSep lexer

reservedOp :: String -> Parser ()
reservedOp = Tok.reservedOp lexer

prefixOp :: String -> (a -> a) -> Op a
prefixOp x f = Ex.Prefix (reservedOp x >> return f)
```

```haskell
table :: [[Op Expr]]
table = [
    [ prefixOp "succ" Succ
    , prefixOp "pred" Pred
    ]
  ]

expr :: Parser Expr
expr = Ex.buildExpressionParser table factor

true, false, zero :: Parser Expr
true  = reserved "true" >> return Tr
false = reserved "false" >> return Fl
zero  = reservedOp "0" >> return Zero

factor :: Parser Expr
factor =
      true
  <|> false
  <|> zero
  <|> parens expr

contents :: Parser a -> Parser a
contents p = do
  Tok.whiteSpace lexer
  r <- p
  eof
  return r

toplevel :: Parser [Expr]
toplevel = semiSep expr

parseExpr :: String -> Either ParseError Expr
parseExpr s = parse (contents expr) "<stdin>" s

parseToplevel :: String -> Either ParseError [Expr]
parseToplevel s = parse (contents toplevel) "<stdin>" s

calcExpr :: String -> Q Exp
calcExpr str = do
  filename <- loc_filename `fmap` location
  case parse (contents expr) filename str of
    Left err -> error (show err)
    Right tag -> [| tag |]

calc :: QuasiQuoter
calc = QuasiQuoter calcExpr err err err
  where err = error "Only defined for values"
```

Testing it out:

```
{-# LANGUAGE QuasiQuotes #-}

import Quasiquote

a :: Expr
a = [calc|true|]
-- Tr

b :: Expr
b = [calc|succ (succ 0)|]
-- Succ (Succ Zero)

c :: Expr
c = [calc|pred (succ 0)|]
-- Pred (Succ Zero)
```

One extremely important feature is the ability to preserve position information so that errors in the embedded language can be traced back to the line of the host syntax.

language-c-quote

Of course since we can provide an arbitrary parser for the quoted expression, one might consider embedding the AST of another language entirely. For example C or CUDA C.

```
hello :: String -> C.Func
hello msg = [cfun|

int main(int argc, const char *argv[])
{
    printf($msg);
    return 0;
}

|]
```

Evaluating this we get back an AST representation of the quoted C program which we can manipulate or print back out to textual C code using `ppr` function.

```
Func
  (DeclSpec [] [] (Tint Nothing))
  (Id "main")
  DeclRoot
  (Params
     [ Param (Just (Id "argc")) (DeclSpec [] [] (Tint Nothing)) DeclRoot
     , Param
         (Just (Id "argv"))
         (DeclSpec [] [ Tconst ] (Tchar Nothing))
         (Array [] NoArraySize (Ptr [] DeclRoot))
```

```
            ]
       False)
    [ BlockStm
        (Exp
           (Just
              (FnCall
                 (Var (Id "printf"))
                 [ Const (StringConst [ "\"Hello Haskell!\"" ] "Hello Haskell!")
                 ])))
    , BlockStm (Return (Just (Const (IntConst "0" Signed 0))))
    ]
```

In this example we just spliced in the anti-quoted Haskell string in the printf statement, but we can pass many other values to and from the quoted expressions including identifiers, numbers, and other quoted expressions which implement the `Lift` type class.

For example now if we wanted programmatically generate the source for a CUDA kernel to run on a GPU we can switch over the CUDA C dialect to emit the C code.

```haskell
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}

import Text.PrettyPrint.Mainland
import qualified Language.C.Syntax as C
import qualified Language.C.Quote.CUDA as Cuda

cuda_fun :: String -> Int -> Float -> C.Func
cuda_fun fn n a = [Cuda.cfun|

__global__ void $id:fn (float *x, float *y) {
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if ( i<$n ) { y[i] = $a*x[i] + y[i]; }
}

|]

cuda_driver :: String -> Int -> C.Func
cuda_driver fn n = [Cuda.cfun|

void driver (float *x, float *y) {
  float *d_x, *d_y;

  cudaMalloc(&d_x, $n*sizeof(float));
  cudaMalloc(&d_y, $n*sizeof(float));

  cudaMemcpy(d_x, x, $n, cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, $n, cudaMemcpyHostToDevice);

  $id:fn<<<($n+255)/256, 256>>>(d_x, d_y);

  cudaFree(d_x);
  cudaFree(d_y);
```

```
    ι εтурι ν;
  }

  |]

makeKernel :: String -> Float -> Int -> [C.Func]
makeKernel fn a n = [
    cuda_fun fn n a
  , cuda_driver fn n
  ]

main :: IO ()
main = do
  let ker = makeKernel "saxpy" 2 65536
  mapM_ (print . ppr) ker
```

Running this we generate:

```
__global__ void saxpy(float* x, float* y)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < 65536) {
        y[i] = 2.0 * x[i] + y[i];
    }
}
int driver(float* x, float* y)
{
    float* d_x, * d_y;

    cudaMalloc(&d_x, 65536 * sizeof(float));
    cudaMalloc(&d_y, 65536 * sizeof(float));
    cudaMemcpy(d_x, x, 65536, cudaMemcpyHostToDevice);
    cudaMemcpy(d_y, y, 65536, cudaMemcpyHostToDevice);
    saxpy<<<(65536 + 255) / 256, 256>>>(d_x, d_y);
    return 0;
}
```

Run the resulting output through `nvcc -ptx -c` to get the PTX associated with the outputted code.

Template Haskell

Of course the most useful case of quasiquotation is the ability to procedurally generate Haskell code itself from inside of Haskell. The `template-haskell` framework provides four entry points for the quotation to generate various types of Haskell declarations and expressions.

Type	Quasiquoted Class	
Q Exp	[e| ... |]	expression
Q Pat	[p| ... |]	pattern
Q Type	[t| ... |]	type
Q [Dec]	[d| ... |]	declaration

```
data QuasiQuoter = QuasiQuoter
  { quoteExp  :: String -> Q Exp
  , quotePat  :: String -> Q Pat
  , quoteType :: String -> Q Type
  , quoteDec  :: String -> Q [Dec]
  }
```

The logic evaluating, splicing, and introspecting compile-time values is embedded within the Q monad, which has a runQ which can be used to evaluate its context. These functions of this monad is deeply embedded in the implementation of GHC.

```
runQ :: Quasi m => Q a -> m a
runIO :: IO a -> Q a
```

Just as before, TemplateHaskell provides the ability to lift Haskell values into the their AST quantities within the quoted expression using the Lift type class.

```
class Lift t where
  lift :: t -> Q Exp

instance Lift Integer where
  lift x = return (LitE (IntegerL x))

instance Lift Int where
  lift x= return (LitE (IntegerL (fromIntegral x)))

instance Lift Char where
  lift x = return (LitE (CharL x))

instance Lift Bool where
  lift True  = return (ConE trueName)
  lift False = return (ConE falseName)

instance Lift a => Lift (Maybe a) where
  lift Nothing  = return (ConE nothingName)
  lift (Just x) = liftM (ConE justName `AppE`) (lift x)

instance Lift a => Lift [a] where
  lift xs = do { xs' <- mapM lift xs; return (ListE xs') }
```

In many cases Template Haskell can be used interactively to explore the AST form of various Haskell syntax.

```
λ: runQ [e| \x -> x |]
LamE [VarP x_2] (VarE x_2)

λ: runQ [d| data Nat = Z | S Nat |]
```

```
[DataD [] Nat_6 [] [Normalc 2_2 [],Normalc 3_1 [(NotStrict,com Nat_6)]] []]

λ: runQ [p| S (S Z)|]
ConP Singleton.S [ConP Singleton.S [ConP Singleton.Z []]]

λ: runQ [t| Int -> [Int] |]
AppT (AppT ArrowT (ConT GHC.Types.Int)) (AppT ListT (ConT GHC.Types.Int))

λ: let g = $(runQ [| \x -> x |])

λ: g 3
3
```

Using <u>Language.Haskell.TH</u> we can piece together Haskell AST element by element but subject to our own custom logic to generate the code. This can be somewhat painful though as the source-language (called ⟨HsSyn⟩) to Haskell is enormous, consisting of around 100 nodes in its AST many of which are dependent on the state of language pragmas.

```
-- builds the function (f = \(a,b) -> a)
f :: Q [Dec]
f = do
  let f = mkName "f"
  a <- newName "a"
  b <- newName "b"
  return [ FunD f [ Clause [TupP [VarP a, VarP b]] (NormalB (VarE a)) [] ] ]
```

```
my_id :: a -> a
my_id x = $( [| x |] )

main = print (my_id "Hello Haskell!")
```

As a debugging tool it is useful to be able to dump the reified information out for a given symbol interactively, to do so there is a simple little hack.

```
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}

import Text.Show.Pretty (ppShow)
import Language.Haskell.TH

introspect :: Name -> Q Exp
introspect n = do
  t <- reify n
  runIO $ putStrLn $ ppShow t
  [| return () |]
```

```
λ: $(introspect 'id)
VarI
   GHC.Base.id
   (ForallT
      [ PlainTV a_1627405383 ]
      []
      (AppT (AppT ArrowT (VarT a_1627405383)) (VarT a_1627405383)))
   Nothing
   (Fixity 9 InfixL)


λ: $(introspect ''Maybe)
TyConI
   (DataD
      []
      Data.Maybe.Maybe
      [ PlainTV a_1627399528 ]
      [ NormalC Data.Maybe.Nothing []
      , NormalC Data.Maybe.Just [ ( NotStrict , VarT a_1627399528 ) ]
      ]
      [])
```

```haskell
import Language.Haskell.TH

foo :: Int -> Int
foo x = x + 1

data Bar

fooInfo :: InfoQ
fooInfo = reify 'foo

barInfo :: InfoQ
barInfo = reify ''Bar
```

```haskell
$( [d| data T = T1 | T2 |] )

main = print [T1, T2]
```

Splices are indicated by `$(f)` syntax for the expression level and at the toplevel simply by invocation of the template Haskell function. Running GHC with `-ddump-splices` shows our code being spliced in at the specific location in the AST at compile-time.

```
$(f)

template_haskell_show.hs:1:1: Splicing declarations
    f
```

```
     ──────/
       template_haskell_show.hs:8:3-10
       f (a_a5bd, b_a5be) = a_a5bd
```

```haskell
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}

module Splice where

import Language.Haskell.TH
import Language.Haskell.TH.Syntax

spliceF :: Q [Dec]
spliceF = do
  let f = mkName "f"
  a <- newName "a"
  b <- newName "b"
  return [ FunD f [ Clause [VarP a, VarP b] (NormalB (VarE a)) [] ] ]

spliceG :: Lift a => a -> Q [Dec]
spliceG n = runQ [d| g a = n |]
```

```haskell
{-# LANGUAGE TemplateHaskell #-}

import Splice

spliceF
spliceG "argument"

main = do
  print $ f 1 2
  print $ g ()
```

At the point of the splice all variables and types used must be in scope, so it must appear after their declarations in the module. As a result we often have to mentally topologically sort our code when using TemplateHaskell such that declarations are defined in order.

See: Template Haskell AST

Antiquotation

Extending our quasiquotation from above now that we have TemplateHaskell machinery we can implement the same class of logic that it uses to pass Haskell values in and pull Haskell values out via pattern matching on templated expressions.

```haskell
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE DeriveDataTypeable #-}
```

```
module Antiquote where

import Data.Generics
import Language.Haskell.TH
import Language.Haskell.TH.Quote

import Text.Parsec
import Text.Parsec.String (Parser)
import Text.Parsec.Language (emptyDef)

import qualified Text.Parsec.Expr as Ex
import qualified Text.Parsec.Token as Tok

data Expr
  = Tr
  | Fl
  | Zero
  | Succ Expr
  | Pred Expr
  | Antiquote String
  deriving (Eq, Show, Data, Typeable)

lexer :: Tok.TokenParser ()
lexer = Tok.makeTokenParser emptyDef

parens :: Parser a -> Parser a
parens = Tok.parens lexer

reserved :: String -> Parser ()
reserved = Tok.reserved lexer

identifier :: Parser String
identifier = Tok.identifier lexer

semiSep :: Parser a -> Parser [a]
semiSep = Tok.semiSep lexer

reservedOp :: String -> Parser ()
reservedOp = Tok.reservedOp lexer

oper s f assoc = Ex.Prefix (reservedOp s >> return f)

table = [ oper "succ" Succ Ex.AssocLeft
        , oper "pred" Pred Ex.AssocLeft
        ]

expr :: Parser Expr
expr = Ex.buildExpressionParser [table] factor

true, false, zero :: Parser Expr
true  = reserved "true" >> return Tr
false = reserved "false" >> return Fl
zero  = reservedOp "0" >> return Zero
```

```haskell
antiquote :: Parser Expr
antiquote = do
  char '$'
  var <- identifier
  return $ Antiquote var

factor :: Parser Expr
factor = true
      <|> false
      <|> zero
      <|> antiquote
      <|> parens expr

contents :: Parser a -> Parser a
contents p = do
  Tok.whiteSpace lexer
  r <- p
  eof
  return r

parseExpr :: String -> Either ParseError Expr
parseExpr s = parse (contents expr) "<stdin>" s


class Expressible a where
  express :: a -> Expr

instance Expressible Expr where
  express = id

instance Expressible Bool where
  express True = Tr
  express False = Fl

instance Expressible Integer where
  express 0 = Zero
  express n = Succ (express (n - 1))


exprE :: String -> Q Exp
exprE s = do
  filename <- loc_filename `fmap` location
  case parse (contents expr) filename s of
    Left err -> error (show err)
    Right exp -> dataToExpQ (const Nothing `extQ` antiExpr) exp

exprP :: String -> Q Pat
exprP s = do
  filename <- loc_filename `fmap` location
  case parse (contents expr) filename s of
    Left err -> error (show err)
    Right exp -> dataToPatQ (const Nothing `extQ` antiExprPat) exp

-- antiquote RHS
```

```haskell
antiExpr :: Expr -> Maybe (Q Exp)
antiExpr (Antiquote v) = Just embed
   where embed = [| express $(varE (mkName v)) |]
antiExpr _ = Nothing

-- antiquote LHS
antiExprPat :: Expr -> Maybe (Q Pat)
antiExprPat (Antiquote v) = Just $ varP (mkName v)
antiExprPat _ = Nothing

mini :: QuasiQuoter
mini = QuasiQuoter exprE exprP undefined undefined
```

```haskell
{-# LANGUAGE QuasiQuotes #-}

import Antiquote

-- extract
a :: Expr -> Expr
a [mini|succ $x|] = x

b :: Expr -> Expr
b [mini|succ $x|] = [mini|pred $x|]

c :: Expressible a => a -> Expr
c x = [mini|succ $x|]

d :: Expr
d = c (8 :: Integer)
-- Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero)))))))

e :: Expr
e = c True
-- Succ Tr
```

Templated Type Families

Just like at the value-level we can construct type-level constructions by piecing together their AST.

```
Type           AST
----------     ----------
t1 -> t2       ArrowT `AppT` t2 `AppT` t2
[t]            ListT `AppT` t
(t1,t2)        TupleT 2 `AppT` t1 `AppT` t2
```

For example consider that type-level arithmetic is still somewhat incomplete in GHC 7.6, but there often cases where the span of typelevel numbers is not full set of integers but is instead some bounded set of numbers. We can instead define operations with a type-family instead of using an inductive definition (which often requires manual proofs) and simply enumerates the entire domain of arguments to the

For example the modulus operator would be non-trivial to implement at type-level but instead we can use the `enumFamily` function to splice in type-family which simply enumerates all possible pairs of numbers up to a desired depth.

```haskell
module EnumFamily where
import Language.Haskell.TH

enumFamily :: (Integer -> Integer -> Integer)
           -> Name
           -> Integer
           -> Q [Dec]
enumFamily f bop upper = return decls
  where
    decls = do
      i <- [1..upper]
      j <- [2..upper]
      return $ TySynInstD bop (rhs i j)

    rhs i j = TySynEqn
      [LitT (NumTyLit i), LitT (NumTyLit j)]
      (LitT (NumTyLit (i `f` j)))
```

```haskell
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TemplateHaskell #-}

import EnumFamily

import Data.Proxy
import GHC.TypeLits

type family Mod (m :: Nat) (n :: Nat) :: Nat
type family Add (m :: Nat) (n :: Nat) :: Nat
type family Pow (m :: Nat) (n :: Nat) :: Nat

enumFamily mod ''Mod 10
enumFamily (+) ''Add 10
enumFamily (^) ''Pow 10

a :: Integer
a = natVal (Proxy :: Proxy (Mod 6 4))
-- 2

b :: Integer
b = natVal (Proxy :: Proxy (Pow 3 (Mod 6 4)))
-- 9

--     enumFamily mod ''Mod 3
--     ======>
--     template_typelevel_splice.hs:7:1-14
```

```
--    type instance Mod 2 1 = 0
--    type instance Mod 2 2 = 0
--    type instance Mod 2 3 = 2
--    type instance Mod 3 1 = 0
--    type instance Mod 3 2 = 1
--    type instance Mod 3 3 = 0
--    ...
```

In practice GHC seems fine with enormous type-family declarations although compile-time may increase a bit as a result.

The singletons library also provides a way to automate this process by letting us write seemingly value-level declarations inside of a quasiquoter and then promoting the logic to the type-level. For example if we wanted to write a value-level and type-level map function for our HList this would normally involve quite a bit of boilerplate, now it can stated very concisely.

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE StandaloneDeriving #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE TypeSynonymInstances #-}

import Data.Singletons
import Data.Singletons.TH

$(promote [d|
  map :: (a -> b) -> [a] -> [b]
  map _ [] = []
  map f (x:xs) = f x : map f xs
  |])

infixr 5 :::

data HList (ts :: [ * ]) where
  Nil :: HList '[]
  (:::) :: t -> HList ts -> HList (t ': ts)

-- TypeLevel
-- MapJust :: [*] -> [Maybe *]
type MapJust xs = Map Maybe xs

-- Value Level
-- mapJust :: [a] -> [Maybe a]
mapJust :: HList xs -> HList (MapJust xs)
mapJust Nil = Nil
mapJust (x ::: xs) = (Just x) ::: mapJust xs
```

```haskell
type A = [Bool, String , Double , ()]

a :: HList A
a = True ::: "foo" ::: 3.14 ::: () ::: Nil


example1 :: HList (MapJust A)
example1 = mapJust a


-- example1 reduces to example2 when expanded
example2 :: HList ([Maybe Bool, Maybe String , Maybe Double , Maybe ()])
example2 = Just True ::: Just "foo" ::: Just 3.14 ::: Just () ::: Nil
```

Templated Type Classes

Probably the most common use of Template Haskell is the automatic generation of type-class instances. Consider if we wanted to write a simple Pretty printing class for a flat data structure that derived the ppr method in terms of the names of the constructors in the AST we could write a simple instance.

```haskell
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FlexibleContexts #-}

module Class where

import Language.Haskell.TH

class Pretty a where
  ppr :: a -> String

normalCons :: Con -> Name
normalCons (NormalC n _) = n

getCons :: Info -> [Name]
getCons cons = case cons of
    TyConI (DataD   _ _ _ tcons _) -> map normalCons tcons
    con -> error $ "Can't derive for:" ++ (show con)

pretty :: Name -> Q [Dec]
pretty dt = do
  info <- reify dt
  Just cls <- lookupTypeName "Pretty"
  let datatypeStr = nameBase dt
  let cons = getCons info
  let dtype = mkName (datatypeStr)
  let mkInstance xs =
        InstanceD
        []                                -- Context
        (AppT
```

```
            (ConT c1s)                        instance
            (ConT dtype))                  -- Head
        [(FunD (mkName "ppr") xs)]         -- Methods
    let methods = map cases cons
    return $ [mkInstance methods]

-- Pattern matches on the ``ppr`` method
cases :: Name -> Clause
cases a = Clause [ConP a []] (NormalB (LitE (StringL (nameBase a)))) []
```

In a separate file invoke the pretty instance at the toplevel, and with `--ddump-splice` if we want to view the spliced class instance.

```
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}

import Class

data PlatonicSolid
  = Tetrahedron
  | Cube
  | Octahedron
  | Dodecahedron
  | Icosahedron

pretty ''PlatonicSolid

main :: IO ()
main = do
  putStrLn (ppr Octahedron)
  putStrLn (ppr Dodecahedron)
```

Templated Singletons

In the previous discussion about singletons, we introduced quite a bit of boilerplate code to work with the singletons. This can be partially abated by using Template Haskell to mechanically generate the instances and classes.

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}

module Singleton where

import Text.Read
import Language.Haskell.TH
import Language.Haskell.TH.Quote
```

```haskell
data Nat = Z | S Nat

data SNat :: Nat -> * where
  SZero :: SNat Z
  SSucc :: SNat n -> SNat (S n)

-- Quasiquoter for Singletons

sval :: String -> Q Exp
sval str = do
  case readEither str of
    Left err -> fail (show err)
    Right n -> do
      Just suc <- lookupValueName "SSucc"
      Just zer <- lookupValueName "SZero"
      return $ foldr AppE (ConE zer) (replicate n (ConE suc))

stype :: String -> Q Type
stype str = do
  case readEither str of
    Left err -> fail (show err)
    Right n -> do
      Just scon <- lookupTypeName "SNat"
      Just suc <- lookupValueName "S"
      Just zer <- lookupValueName "Z"
      let nat = foldr AppT (PromotedT zer) (replicate n (PromotedT suc))
      return $ AppT (ConT scon) nat

spat :: String -> Q Pat
spat str = do
  case readEither str of
    Left err -> fail (show err)
    Right n -> do
      Just suc <- lookupValueName "SSucc"
      Just zer <- lookupValueName "SZero"
      return $ foldr (\x y -> ConP x [y]) (ConP zer []) (replicate n (suc))

sdecl :: String -> a
sdecl _ = error "Cannot make toplevel declaration for snat."

snat :: QuasiQuoter
snat = QuasiQuoter sval spat stype sdecl
```

Trying it out by splicing code at the expression level, type level and as patterns.

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TemplateHaskell #-}

import Singleton
```

```haskell
zero :: [snat|0|]
zero =  [snat|0|]

one :: [snat|1|]
one =  [snat|1|]

two :: [snat|2|]
two =  [snat|2|]

three :: [snat|3|]
three  = [snat|3|]

test :: SNat a -> Int
test x = case x of
  [snat|0|] -> 0
  [snat|1|] -> 1
  [snat|2|] -> 2
  [snat|3|] -> 3

isZero :: SNat a -> Bool
isZero [snat|0|] = True
isZero _ = False
```

The singletons package takes this idea to its logical conclusion allow us to toplevel declarations of seemingly regular Haskell syntax with singletons spliced in, the end result resembles the constructions in a dependently typed language if one squints hard enough.

```haskell
{-# LANGUAGE GADTs #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE QuasiQuotes #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE TypeOperators #-}

import Data.Singletons
import Data.Singletons.TH

$(singletons [d|
  data Nat = Zero | Succ Nat
    deriving (Eq, Show)

  plus :: Nat -> Nat -> Nat
  plus Zero     n = n
  plus (Succ m) n = Succ (plus m n)

  isEven :: Nat -> Bool
  isEven Zero = True
  isEven (Succ Zero) = False
  isEven (Succ (Succ n)) = isEven n
```

```
⌋⌋)
```

After template splicing we see that we now that several new constructs in scope:

```haskell
type SNat a = Sing Nat a

type family IsEven a :: Bool
type family Plus a b :: Nat

sIsEven :: Sing Nat t0 -> Sing Bool (IsEven t0)
splus   :: Sing Nat a -> Sing Nat b -> Sing Nat (Plus a b)
```

Lenses

There are several implementations of note that are mostly compatible but differ in scope:

- *lens-family-core*
- *fc-labels*
- *data-lens-light*
- *lens*

Should I use lens library?

The `lens` library is considered by some Haskellers to be pathological in the sense that it doesn't conform to many of the common conventions present in the bulk of the community. Some care should taken when considering its use. lens is effectively a laboratory for a certain set of emerging ideas.

By analogy, lens is most readily comparable to the "Boost C++" of the Haskell world. A library which seemingly offers a solution to an enormous number of problems, at the cost of bringing in an enormous edifice of code and different way of working that can be polarizing to certain practitioners.

Van Laarhoven Lenses

At its core a lens is a form of coupled getter and setter functions as a value under an existential functor.

```haskell
--         +---- a : Type of structure
--         | +-- b : Type of target
--         | |
type Lens' a b = forall f. Functor f => (b -> f b) -> (a -> f a)
```

There are two derivations of Van Laarhoven lenses: one that allows polymorphic update, and one that is strictly monomorphic. Let's just consider the monomorphic variant first:

```haskell
type Lens' a b = forall f. Functor f => (b -> f b) -> (a -> f a)

newtype Const x a  = Const { runConst :: x } deriving Functor
newtype Identity a = Identity { runIdentity :: a } deriving Functor

lens :: (a -> b) -> (a -> b -> a) -> Lens' a b
lens getter setter l a = setter a <$> l (getter a)
```

```haskell
set :: Lens' a b -> b -> a -> a
set l b = runIdentity . l (const (Identity b))

get :: Lens' a b -> a -> b
get l = runConst . l Const

over :: Lens' a b -> (b -> b) -> a -> a
over l f a = set l (f (get l a)) a
```

```haskell
infixl 1 &
infixr 4 .~
infixr 4 %~
infixl 8 ^.

(&) :: a -> (a -> b) -> b
(&) = flip ($)

(^.) :: a -> Lens' a b -> b
(^.) = flip get

(.~) :: Lens' a b -> b -> a -> a
(.~) = set

(%~) :: Lens' a b -> (b -> b) -> a -> a
(%~) = over
```

Such that we have:

```haskell
a ^. (lens getter setter)        -- getter a
a  & (lens getter setter) .~ b   -- setter a b
```

Law 1

```haskell
get l (set l b a) = b
```

Law 2

```haskell
set l (get l a) a = a
```

Law 3

```haskell
set l b1 (set l b2 a) = set l b1 a
```

With composition identities:

```
x^.a.b ≡ x^.a^.b
a.b %~ f ≡ a %~ b %~ f

x ^. id ≡ x
id %~ f ≡ f
```

While this may look like a somewhat convoluted way of reinventing record update, consider that the types of these functions align very nicely. Lenses can be composed using the normal `(.)` composition, although in the "reverse" or "opposite" direction of function composition.

```
f      :: a -> b
g      :: b -> c
g . f :: a -> c

f      :: Lens a b  ~  (b -> f b) -> (a -> f a)
g      :: Lens b c  ~  (c -> f c) -> (b -> f b)
f . g :: Lens a c  ~  (c -> f c) -> (a -> f a)
```

```haskell
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE NoMonomorphismRestriction #-}

import Data.Functor

type Lens' a b = forall f. Functor f => (b -> f b) -> (a -> f a)

newtype Const x a = Const { runConst :: x } deriving Functor
newtype Identity a = Identity { runIdentity :: a } deriving Functor

lens :: (a -> b) -> (a -> b -> a) -> Lens' a b
lens getter setter f a = fmap (setter a) (f (getter a))

set :: Lens' a b -> b -> a -> a
set l b = runIdentity . l (const (Identity b))

view :: Lens' a b -> a -> b
view l = runConst . l Const

over :: Lens' a b -> (b -> b) -> a -> a
over l f a = set l (f (view l a)) a

compose :: Lens' a b -> Lens' b c -> Lens' a c
compose l s = l . s

id' :: Lens' a a
```

```
iu = iu

infixl 1 &
infixr 4 .~
infixr 4 %~
infixl 8 ^.

(^.) :: a -> Lens' a b -> b
(^.) = flip view

(.~) :: Lens' a b -> b -> a -> a
(.~) = set

(%~) :: Lens' a b -> (b -> b) -> a -> a
(%~) = over

(&) :: a -> (a -> b) -> b
(&) = flip ($)

(+~), (-~), (*~) :: Num b => Lens' a b -> b -> a -> a
f +~ b = f %~ (+b)
f -~ b = f %~ (subtract b)
f *~ b = f %~ (*b)

-- Usage

data Foo = Foo { _a :: Int } deriving Show
data Bar = Bar { _b :: Foo } deriving Show

a :: Lens' Foo Int
a = lens getter setter
  where
    getter :: Foo -> Int
    getter = _a

    setter :: Foo -> Int -> Foo
    setter = (\f new -> f { _a = new })


b :: Lens' Bar Foo
b = lens getter setter
  where
    getter :: Bar -> Foo
    getter = _b

    setter :: Bar -> Foo -> Bar
    setter = (\f new -> f { _b = new })

foo :: Foo
foo = Foo 3

bar :: Bar
bar = Bar foo
```

```haskell
example1  = view a foo
example2 = set a 1 foo
example3 = over a (+1) foo
example4 = view (b `compose` a) bar

example1' = foo  ^. a
example2' = foo  &  a .~ 1
example3' = foo  &  a %~ (+1)
example4' = bar  ^. b . a
```

It turns out that these simple ideas lead to a very rich set of composite combinators that be used to work with substructures of complex data structures.

Combinator Description

`view`	View a single target or fold the targets of a monoidal quantity.
`set`	Replace target with a value and return updated structure.
`over`	Update targets with a function and return updated structure.
`to`	Construct a retrieval function from an arbitrary Haskell function.
`traverse`	Map each element of a structure to an action and collect results.
`ix`	Target the given index of a generic indexable structure.
`toListOf`	Return a list of the targets.
`firstOf`	Returns `Just` the target of a prism or Nothing.

Certain patterns show up so frequently that they warrant their own operators, although they can be expressed with textual terms as well.

Symbolic Textual Equivalent Description

Symbolic	Textual Equivalent	Description
`^.`	`view`	Access value of target
`.~`	`set`	Replace target `x`
`%~`	`over`	Apply function to target
`+~`	`over t (+n)`	Add to target
`-~`	`over t (-n)`	Subtract to target
`*~`	`over t (*n)`	Multiply to target
`//~`	`over t (//n)`	Divide to target
`^~`	`over t (^n)`	Integral power to target
`^^~`	`over t (^^n)`	Fractional power to target
`||~`	`over t (|| p)`	Logical or to target
`&&~`	`over t (&& p)`	Logical and to target
`<>~`	`over t (<> n)`	Append to a monoidal target
`?~`	`set t (Just x)`	Replace target with `Just x`
`^?`	`firstOf`	Return `Just` target or `Nothing`
`^..`	`toListOf`	View list of targets

Constructing the lens field types from an arbitrary datatype involves a bit of boilerplate code generation, but compiles into simple calls which translate the fields of a record into functions involving the `lens` function and logic for the getter and the setter.

```haskell
import Control.Lens

data Foo = Foo { _field :: Int }

field :: Lens' Foo Int
```

```haskell
field = lens getter setter
  where
    getter :: Foo -> Int
    getter = _field

    setter :: Foo -> Int -> Foo
    setter = (\f new -> f { _field = new })
```

These are pure boilerplate, and Template Haskell can automatically generate these functions using `makeLenses` by introspecting the AST at compile-time.

```haskell
{-# LANGUAGE TemplateHaskell #-}

import Control.Lens

data Foo = Foo { _field :: Int } deriving Show
makeLenses ''Foo
```

The simplest usage of lens is simply as a more compositional way of dealing with record access and updates, shown below in comparison with traditional record syntax:

```haskell
{-# LANGUAGE TemplateHaskell #-}

import Control.Lens

data Rec = MkRec { _foo :: Int , _bar :: Int } deriving Show
makeLenses ''Rec

x :: Rec
x = MkRec { _foo = 1024, _bar = 1024 }

get1 :: Int
get1 = (_foo x) + (_bar x)

get2 :: Int
get2 = (x ^. foo) + (x ^. bar)

get3 :: Int
get3 = (view foo x) + (view bar x)


set1 :: Rec
set1 = x { _foo = 1, _bar = 2 }

set2 :: Rec
set2 = x & (foo .~ 1) . (bar .~ 2)

set3 :: Rec
set3 = x & (set foo 1) . (set bar 2)
```

This pattern has great utility when it comes when dealing with complex and deeply nested structures:

```haskell
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE NoMonomorphismRestriction #-}
{-# LANGUAGE RankNTypes #-}


import Control.Lens
import Control.Lens.TH

data Record1 = Record1
  { _a :: Int
  , _b :: Maybe Record2
  } deriving Show

data Record2 = Record2
  { _c :: String
  , _d :: [Int]
  } deriving Show

makeLenses ''Record1
makeLenses ''Record2

records :: [Record1]
records = [
    Record1 {
      _a = 1,
      _b = Nothing
    },
    Record1 {
      _a = 2,
      _b = Just $ Record2 {
        _c = "Picard",
        _d = [1,2,3]
      }
    },
    Record1 {
      _a = 3,
      _b = Just $ Record2 {
        _c = "Riker",
        _d = [4,5,6]
      }
    },
    Record1 {
      _a = 4,
      _b = Just $ Record2 {
        _c = "Data",
        _d = [7,8,9]
      }
    }
  ]
```

```
-- Lens targets
ids       = traverse.a
names     = traverse.b._Just.c
nums      = traverse.b._Just.d
listn n   = traverse.b._Just.d.ix n

-- Modify to set all 'id' fields to 0
ex1 :: [Record1]
ex1 = set ids 0 records

-- Return a view of the concatenated 'd' fields for all nested records.
ex2 :: [Int]
ex2 = view nums records
-- [1,2,3,4,5,6,7,8,9]

-- Increment all 'id' fields by 1
ex3 :: [Record1]
ex3 = over ids (+1) records

-- Return a list of all 'c' fields.
ex4 :: [String]
ex4 = toListOf names records
-- ["Picard","Riker","Data"]

-- Return the the second element of all 'd' fields.
ex5 :: [Int]
ex5 = toListOf (listn 2) records
-- [3,6,9]
```

Lens also provides us with an optional dense slurry of operators that expand into combinations of the core combinators. Many of the operators do have a <u>consistent naming scheme</u>.

The sheer number of operators provided by lens is staggering for some, but all of the operators can be written in terms of the textual functions ((set) , (view) , (over) , (at) , ...) and some people prefer to use these instead.

If one buys into lens model, it can serve as a partial foundation to write logic over a wide variety of data structures and computations and subsume many of the existing patterns found in the Prelude.

```
{-# LANGUAGE NoMonomorphismRestriction #-}

import Control.Lens
import Numeric.Lens
import Data.Complex.Lens

import Data.Complex
import qualified Data.Map as Map

l :: Num a => a
l = view _1 (100, 200)
-- 100

m :: Num a => (a, a, a)
```

```
m = (100,200,200) & _3 %~ (+100)
-- (100,200,300)

n :: Num a => [a]
n = [100,200,300] & traverse +~ 1
-- [101,201,301]

o :: Char
o = "frodo" ^?! ix 3
-- 'd'

p :: Num a => [a]
p = [[1,2,3], [4,5,6]] ^. traverse
-- [1,2,3,4,5,6]

q :: Num a => [a]
q = [1,2,3,4,5] ^. _tail
-- [2,3,4,5]

r :: Num a => [Maybe a]
r = [Just 1, Just 2, Just 3] & traverse._Just +~ 1
-- [Just 2, Just 3, Just 4]

s :: Maybe String
s = Map.fromList [("foo", "bar")] ^. at "foo"
-- Just "bar"

t :: Integral a => Maybe a
t = "1010110" ^? binary
-- Just 86

u :: Complex Float
u = (mkPolar 1 pi/2) & _phase +~ pi
-- 0.5 :+ 8.742278e-8

v :: [Integer]
v = [1..10] ^.. folded.filtered even
-- [2,4,6,8,10]

w :: [Integer]
w = [1, 2, 3, 4] & each . filtered even *~ 10
-- [1, 20, 3, 40]

x :: Num a => Maybe a
x = Left 3 ^? _Left
-- Just 3
```

See:

- A Little Lens Tutorial
- CPS based functional references
- Lens infix operators

lens family

The interface for `lens-family` is very similar to `lens` but with a smaller API and core.

```haskell
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE NoMonomorphismRestriction #-}

import Lens.Family
import Lens.Family.TH
import Lens.Family.Stock
import Data.Traversable

data Record1 = Record1
  { _a :: Int
  , _b :: Maybe Record2
  } deriving Show

data Record2 = Record2
  { _c :: String
  , _d :: [Int]
  } deriving Show

mkLenses ''Record1
mkLenses ''Record2

records :: [Record1]
records = [
    Record1 {
      _a = 1,
      _b = Nothing
    },
    Record1 {
      _a = 2,
      _b = Just $ Record2 {
        _c = "Picard",
        _d = [1,2,3]
      }
    },
    Record1 {
      _a = 3,
      _b = Just $ Record2 {
        _c = "Riker",
        _d = [4,5,6]
      }
    },
    Record1 {
      _a = 4,
      _b = Just $ Record2 {
        _c = "Data",
        _d = [7,8,9]
      }
    }
  ]
```

```
ids   = traverse.a
names = traverse.b._Just.c
nums  = traverse.b._Just.d

ex1 = set ids 0 records
ex2 = view nums records
ex3 = over ids (+1) records
ex4 = toListOf names records
```

Polymorphic Update

```
--          +---- a  : Type of input structure
--          | +-- a' : Type of output structure
--          | |
type Lens a a' b b' = forall f. Functor f => (b -> f b') -> (a -> f a')
--              | |
--              | +-- b  : Type of input target
--              +---- b' : Type of output target
```

```
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE NoMonomorphismRestriction #-}

import Data.Functor

type Lens a a' b b' = forall f. Functor f => (b -> f b') -> (a -> f a')
type Lens' a b = Lens a a b b

newtype Const x a  = Const { runConst :: x } deriving Functor
newtype Identity a = Identity { runIdentity :: a } deriving Functor

lens :: (a -> b) -> (a -> b' -> a') -> Lens a a' b b'
lens getter setter f a = fmap (setter a) (f (getter a))

set :: Lens a a' b b' -> b' -> a -> a'
set l b = runIdentity . l (const (Identity b))

get :: Lens a a' b b' -> a -> b
get l = runConst . l Const

over :: Lens a a' b b' -> (b -> b') -> a -> a'
over l f a = set l (f (get l a)) a

compose :: Lens a a' b b' -> Lens b b' c c' -> Lens a a' c c'
compose l s = l . s

id' :: Lens a a a a
id' = id
```

```haskell
infixl 1 &
infixr 4 .~
infixr 4 %~
infixr 8 ^.

(^.) = flip get
(.~) = set
(%~) = over

(&) :: a -> (a -> b) -> b
(&) = flip ($)

(+~), (-~), (*~) :: Num b => Lens a a b b -> b -> a -> a
f +~ b = f %~ (+b)
f -~ b = f %~ (subtract b)
f *~ b = f %~ (*b)

-- Monomorphic Update
data Foo = Foo { _a :: Int } deriving Show
data Bar = Bar { _b :: Foo } deriving Show

a :: Lens' Foo Int
a = lens getter setter
  where
    getter :: Foo -> Int
    getter = _a

    setter :: Foo -> Int -> Foo
    setter = (\f new -> f { _a = new })

b :: Lens' Bar Foo
b = lens getter setter
  where
    getter :: Bar -> Foo
    getter = _b

    setter :: Bar -> Foo -> Bar
    setter = (\f new -> f { _b = new })

-- Polymorphic Update
data Pair a b = Pair a b deriving Show

pair :: Pair Int Char
pair = Pair 1 'b'

_1 :: Lens (Pair a b) (Pair a' b) a a'
_1 f (Pair a b) = (\x -> Pair x b) <$> f a

_2 :: Lens (Pair a b) (Pair a b') b b'
_2 f (Pair a b) = (\x -> Pair a x) <$> f b

ex1 = pair ^. _1
ex2 = pair ^. _2
```

```
ex5 = pair & _1 . a
ex4 = pair & (_1  %~ (+1))
            . (_2  .~ 1)
```

State and Zoom

Within the context of the state monad there are a particularly useful set of lens patterns.

- `use` - View a target from the state of the State monad.
- `assign` - Replace the target within a State monad.
- `zoom` - Modify a target of the state with a function and perform it on the global state of the State monad.

So, for example, if we wanted to write a little physics simulation of the random motion of particles in a box, we could use the `zoom` function to modify the state of our particles in each step of the simulation.

```haskell
{-# LANGUAGE TemplateHaskell #-}

import Control.Lens
import Control.Monad.State
import System.Random

data Vector = Vector
    { _x :: Double
    , _y :: Double
    } deriving (Show)

data Box = Box
    { _particles :: [Particle]
    } deriving (Show)

data Particle = Particle
    { _pos :: Vector
    , _vel :: Vector
    } deriving (Show)

makeLenses ''Box
makeLenses ''Particle
makeLenses ''Vector

step :: StateT Box IO ()
step = zoom (particles.traverse) $ do
    dx <- use (vel.x)
    dy <- use (vel.y)
    pos.x += dx
    pos.y += dy

particle :: IO Particle
particle = do
  vx <- randomIO
  vy <- randomIO
  return $ Particle (Vector 0 0) (Vector vx vy)
```

```haskell
simulate :: IO Box
simulate = do
  particles <- replicateM 5 particle
  let simulation = replicateM 5 step
  let box = Box particles
  execStateT simulation box

main :: IO ()
main = simulate >>= print
```

This results in a final state like the following.

```
Box
  { _particles =
    [ Particle
        { _pos =
            Vector { _x = 3.268546939011934 , _y = 4.356638656040016 }
        , _vel =
            Vector { _x = 0.6537093878023869 , _y = 0.8713277312080032 }
        }
    , Particle
        { _pos =
            Vector { _x = 0.5492296641559635 , _y = 0.27244422070641594 }
        , _vel =
            Vector { _x = 0.1098459328311927 , _y = 5.448884414128319e-2 }
        }
    , Particle
        { _pos =
            Vector { _x = 3.961168796078436 , _y = 4.9317543172941765 }
        , _vel =
            Vector { _x = 0.7922337592156872 , _y = 0.9863508634588353 }
        }
    , Particle
        { _pos =
            Vector { _x = 4.821390854065674 , _y = 1.6601909953629823 }
        , _vel =
            Vector { _x = 0.9642781708131349 , _y = 0.33203819907259646 }
        }
    , Particle
        { _pos =
            Vector { _x = 2.6468253761062943 , _y = 2.161403445396069 }
        , _vel =
            Vector { _x = 0.5293650752212589 , _y = 0.4322806890792138 }
        }
    ]
  }
```

Lens + Aeson

One of the best showcases for lens is writing transformations over arbitrary JSON structures. For

example consider some sample data related to Kiva loans.

```json
{
    "loans":[
        {
            "id":2930,
            "terms":{
                "local_payments":[
                    {
                        "due_date":"2007-02-08T08:00:00Z",
                        "amount":13.75
                    },
                    {
                        "due_date":"2007-03-08T08:00:00Z",
                        "amount":93.75
                    },
                    {
                        "due_date":"2007-04-08T07:00:00Z",
                        "amount":43.75
                    },
                    {
                        "due_date":"2007-05-08T07:00:00Z",
                        "amount":63.75
                    },
                    {
                        "due_date":"2007-06-08T07:00:00Z",
                        "amount":93.75
                    },
                    {
                        "due_date":"2007-07-08T05:00:00Z",
                        "amount": null
                    },
                    {
                        "due_date":"2007-07-08T07:00:00Z",
                        "amount":93.75
                    },
                    {
                        "due_date":"2007-08-08T07:00:00Z",
                        "amount":93.75
                    },
                    {
                        "due_date":"2007-09-08T07:00:00Z",
                        "amount":93.75
                    }
                ]
            }
        }
    ]
}
```

Then using `Data.Aeson.Lens` we can traverse the structure using our lens combinators.

```haskell
{-# LANGUAGE OverloadedStrings #-}

import Control.Lens

import Data.Aeson.Lens
import Data.Aeson (decode, Value)
import Data.ByteString.Lazy as BL

main :: IO ()
main = do
  contents <- BL.readFile "kiva.json"
  let Just json = decode contents :: Maybe Value

  let vals :: [Double]
      vals = json ^.. key "loans"
                     . values
                     . key "terms"
                     . key "local_payments"
                     . values
                     . key "amount"
                     . _Double
  print vals
```

```
[13.75,93.75,43.75,63.75,93.75,93.75,93.75,93.75]
```

Categories

Alas we come to the topic of category theory. Some might say all discussion of Haskell eventually leads here at one point or another.

Nevertheless the overall importance of category theory in the context of Haskell has been somewhat overstated and unfortunately mystified to some extent. The reality is that amount of category theory which is directly applicable to Haskell roughly amounts to a subset of the first chapter of any undergraduate text.

Algebraic Relations

Grossly speaking category theory is not terribly important to Haskell programming, and although some libraries derive some inspiration from the subject most do not. What is more important is a general understanding of equational reasoning and a familiarity with various algebraic relations.

Certain relations show up so frequently we typically refer to their properties by name (often drawn from an equivalent abstract algebra concept). Consider a binary operation `a `op` b` .

Associativity

```
a `op` (b `op` c) = (a `op` b) `op` c
```

Commutativity

```
a `op` b = b `op` a
```

Units

```
a `op` e = a
e `op` a = a
```

Inversion

```
(inv a) `op` a = e
a `op` (inv a) = e
```

Zeros

```
a `op` e = e
e `op` a = e
```

Linearity

```
f (x `op` y) = f x `op` f y
```

Idempotency

```
f (f x) = f x
```

Distributivity

```
a `f` (b `g` c) = (a `f` b) `g` (a `f` c)
(b `g` c) `f` a = (b `f` a) `g` (c `f` a)
```

Anticommutativity

```
a `op` b = inv (b `op` a)
```

And of course combinations of these properties over multiple functions gives rise to higher order systems of relations that occur over and over again throughout functional programming, and once we recognize them we can abstract over them. For instance a monoid is a combination of a unit and a

Categories

The most basic structure is a category which is an algebraic structure of objects (Obj) and morphisms (Hom) with the structure that morphisms compose associatively and the existence of an identity morphism for each object.

With kind polymorphism enabled we can write down the general category parameterized by a type variable "c" for category, and the instance Hask the category of Haskell types with functions between types as morphisms.

```haskell
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE TypeSynonymInstances #-}

import Prelude hiding ((.), id)

-- Morphisms
type (a ~> b) c = c a b

class Category (c :: k -> k -> *) where
  id :: (a ~> a) c
  (.) :: (y ~> z) c -> (x ~> y) c -> (x ~> z) c

type Hask = (->)

instance Category Hask where
  id x = x
  (f . g) x = f (g x)
```

Isomorphisms

Two objects of a category are said to be isomorphic if we can construct a morphism with 2-sided inverse that takes the structure of an object to another form and back to itself when inverted.

```haskell
f  :: a -> b
f' :: b -> a
```

Such that:

```haskell
f . f' = id
f'. f  = id
```

For example the types Either () a and Maybe a are isomorphic.

```haskell
{-# LANGUAGE ExplicitForAll #-}
```

```haskell
data Iso a b = Iso { to :: a -> b, from :: b -> a }

f :: forall a. Maybe a -> Either () a
f (Just a) = Right a
f Nothing  = Left ()

f' :: forall a. Either () a -> Maybe a
f' (Left _)  = Nothing
f' (Right a) = Just a

iso :: Iso (Maybe a) (Either () a)
iso = Iso f f'

data V = V deriving Eq

ex1 = f  (f' (Right V)) == Right V
ex2 = f' (f  (Just V))  == Just V
```

```haskell
data Iso a b = Iso { to :: a -> b, from :: b -> a }

instance Category Iso where
  id = Iso id id
  (Iso f f') . (Iso g g') = Iso (f . g) (g' . f')
```

Duality

One of the central ideas is the notion of duality, that reversing some internal structure yields a new structure with a "mirror" set of theorems. The dual of a category reverse the direction of the morphisms forming the category C^{Op}.

```haskell
import Control.Category
import Prelude hiding ((.), id)

newtype Op a b = Op (b -> a)

instance Category Op where
  id = Op id
  (Op f) . (Op g) = Op (g . f)
```

See:

Functors

Functors are mappings between the objects and morphisms of categories that preserve identities and composition.

```haskell
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeSynonymInstances #-}

import Prelude hiding (Functor, fmap, id)

class (Category c, Category d) => Functor c d t where
  fmap :: c a b -> d (t a) (t b)

type Hask = (->)

instance Category Hask where
  id x = x
  (f . g) x = f (g x)

instance Functor Hask Hask [] where
  fmap f [] = []
  fmap f (x:xs) = f x : (fmap f xs)
```

```
fmap id ≡ id
fmap (a . b) ≡ (fmap a) . (fmap b)
```

Natural Transformations

Natural transformations are mappings between functors that are invariant under interchange of morphism composition order.

```haskell
type Nat f g = forall a. f a -> g a
```

Such that for a natural transformation (h) we have:

```
fmap f . h ≡ h . fmap f
```

The simplest example is between (f = List) and (g = Maybe) types.

```haskell
headMay :: forall a. [a] -> Maybe a
headMay []     = Nothing
headMay (x:xs) = Just x
```

Regardless of how we chase (safeHead), we end up with the same result.

```
fmap f (headMay xs) ≡ headMay (fmap f xs)
```

```
fmap f (headMay [])
= fmap f Nothing
= Nothing

headMay (fmap f [])
= headMay []
= Nothing
```

```
fmap f (headMay (x:xs))
= fmap f (Just x)
= Just (f x)

headMay (fmap f (x:xs))
= headMay [f x]
= Just (f x)
```

Or consider the Functor (->) .

```
f :: (Functor t)
  => (->) a b
  -> (->) (t a) (t b)
f = fmap

g :: (b -> c)
  -> (->) a b
  -> (->) a c
g = (.)

c :: (Functor t)
  => (b -> c)
  -> (->) (t a) (t b)
  -> (->) (t a) (t c)
c = f . g
```

```
f . g x = c x . g
```

A lot of the expressive power of Haskell types comes from the interesting fact that, with a few caveats, polymorphic Haskell functions are natural transformations.

See: You Could Have Defined Natural Transformations

Yoneda Lemma

The Yoneda lemma is an elementary, but deep result in Category theory. The Yoneda lemma states that for any functor F , the types F a and ∀ b. (a -> b) -> F b are isomorphic.

```haskell
{-# LANGUAGE RankNTypes #-}

embed :: Functor f => f a -> (forall b . (a -> b) -> f b)
embed x f = fmap f x

unembed :: Functor f => (forall b . (a -> b) -> f b) -> f a
unembed f = f id
```

So that we have:

```haskell
embed . unembed ≡ id
unembed . embed ≡ id
```

The most broad hand-wavy statement of the theorem is that an object in a category can be represented by the set of morphisms into it, and that the information about these morphisms alone sufficiently determines all properties of the object itself.

In terms of Haskell types, given a fixed type (a) and a functor (f), if we have some a higher order polymorphic function (g) that when given a function of type $(a \to b)$ yields $(f\ b)$ then the behavior (g) is entirely determined by $(a \to b)$ and the behavior of (g) can written purely in terms of $(f\ a)$.

See:

- Reverse Engineering Machines with the Yoneda Lemma

Kleisli Category

Kleisli composition (i.e. Kleisli Fish) is defined to be:

```haskell
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
f >=> g ≡ \x -> f x >>= g

(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> a -> m c
(<=<) = flip (>=>)
```

The monad laws stated in terms of the Kleisli category of a monad (m) are stated much more symmetrically as one associativity law and two identity laws.

```haskell
(f >=> g) >=> h ≡ f >=> (g >=> h)
return >=> f ≡ f
f >=> return ≡  f
```

Stated simply that the monad laws above are just the category laws in the Kleisli category.

```haskell
{-# LANGUAGE TypeOperators #-}
```

```
{-# LANGUAGE ExplicitForAll #-}

import Control.Monad
import Control.Category
import Prelude hiding ((.))

-- Kleisli category
newtype Kleisli m a b = K (a -> m b)

-- Kleisli morphisms ( a -> m b )
type (a :~> b) m = Kleisli m a b

instance Monad m => Category (Kleisli m) where
  id            = K return
  (K f) . (K g) = K (f <=< g)


just :: (a :~> a) Maybe
just = K Just

left :: forall a b. (a :~> b) Maybe -> (a :~> b) Maybe
left f = just . f

right :: forall a b. (a :~> b) Maybe -> (a :~> b) Maybe
right f = f . just
```

For example, `Just` is just an identity morphism in the Kleisli category of the `Maybe` monad.

```
Just >=> f ≡ f
f >=> Just ≡ f
```

Resources

- Category Theory, Awodey
- Category Theory Foundations
- The Catsters