The Monad.Reader Issue 13

by Chris Eidhof ⟨ce@tupil.com⟩
and Derek Elkins ⟨derek.a.elkins@gmail.com⟩
and Stephen Hicks ⟨sdh33@cornell.edu⟩
and Eelco Lempsink ⟨eml@tupil.com⟩
and Brent Yorgey ⟨byorgey@cis.upenn.edu⟩

12 March, 2009

Wouter Swierstra, editor.

Contents

Editorial

by Wouter Swierstra ⟨wouter@chalmers.se⟩

A lot has happened since the last release of The Monad.Reader. The financial markets have crashed. One important reason seems to be the trading in financial derivatives. These derivatives can be so complex that it's very hard to estimate their value. Now if only there was a domain specific language for describing and evaluating such financial contracts...

A few weeks ago, GMail had a major outage. Edwin Brady pointed out a press release by Google that blamed 'unexpected side effects of some new code' – now if only there was some way to ensure code doesn't have side effects...

This leads me to believe that Haskell is currently solving problems that will affect society in ten years or so. I predict there's a major telecom blackout because Erlang fails to do parallel garbage collection. Maybe major bank systems will crash because of a mistake in taking the wrong locks during transactions. Or perhaps an epic fail of Amazon's databases after an incorrect type cast.

In the meantime, enjoy this issue of The Monad.Reader: Stephen Hicks provides an account of his award winning ICFP Programming Contest entry; Brent Yorgey gives an overview of Haskell's type class libraries; Chris Eidhof and Eelco Lempsink have written a review of **Real World Haskell**; and finally, Derek Elkins explains some of the category theory underlying Haskell's standard monads. Whether you're into category theory or TEX hacking, I'm sure there's something in this issue you'll enjoy.

Rapid Prototyping in TEX

by Stephen Hicks ⟨sdh33@cornell.edu⟩

This is a brief report of my experience using TEX in the 2008 ICFP programming contest, winning the Judges' Prize for my submission as The Lone TEXnician. I provide cleaned-up code fragments for a few of the more interesting routines.

Why TEX?

There were several factors that motivated my choice of TEX for this contest. Firstly, I spent several weeks the previous spring rewriting a large part of the TEX output routine to better automate placement of margin notes (you can see the problem for yourself by starting a long `\marginpar` at the bottom of a page), and as a result I found myself very much in the mindset that "when all you've got is a typesetter, everything starts to look like a document." In addition, my efforts of recruiting a team for the contest this year failed miserably, and when faced with the task of programming a sufficiently smart pathfinding algorithm on my own, I decided I didn't stand a chance against the teams of Java programmers that seemed to actually know what they were doing. It soon became clear that my best chance was to see how far I could get with a solution in TEX.

Problem statement

The basic problem was to write a program to control a Mars rover. The program connects to a network server, and then listens for messages and sends instructions to navigate the rover home. The server first sends an initialization message (`I` ⟨D_x⟩ ⟨D_y⟩ ⟨*time limit*⟩ ⟨*min sensor*⟩ ⟨*max sensor*⟩ ⟨*max speed*⟩ ⟨*max soft turn speed*⟩ ⟨*max hard turn speed*⟩ `;`) giving a bunch of information that we mostly ignore (we do save the max speeds, though). From then on, every 100 milliseconds it sends a telemetry message (`T` ⟨*time*⟩ ⟨*state*⟩ ⟨x⟩ ⟨y⟩ ⟨*direction*⟩ ⟨*speed*⟩ ⟨*obstacles...*⟩ `;`)

telling us where we are, how fast we're going and in what direction, and what obstacles we see around us: boulders, which we bounce off of; craters, which we fall into and die; and martians, which chase us and disassemble us if we're caught. Finally, there are also messages for special events such as failure (hitting a crater or martian, or running out of time), success, or bouncing off a boulder. The rover's state consists of a 5-way turning state (turning hard left, turning soft left, straight, turning soft right, or turning hard right) and a 3-way acceleration state (braking, coasting, or accelerating). The program sends single-character instructions (`a`, `b`, `l`, and `r`) to the server to change the state incrementally, so that it takes three `r` instructions to go from turning hard left to soft right (as well as an unknown angular acceleration time). For more details, see the contest archives [1].

Basic strategy

A more narrative account of the choices made during the development process, as well as the complete source code from the entry, can be found in the related blog post [2]. I will therefore discuss here only the final results and give a few illustrative code fragments, which have been edited for clarity.

Network access

We need to figure out how to let TEX interact with the outside world. To this end, I was inspired by PerlTEX, which allows running arbitrary Perl code from within TEX by launching a TEX process from Perl, and then monitoring stdout for blocks of code, the results of which are sent back through stdin each time TEX asks for input. Listing 1 is a mostly self-contained fragment to illustrate how this works. The process is depicted graphically in Figure 1.

In this fragment we define four macros. The first, `\send`, prints a message to stdout for Perl to send to the server. The other three are used for parsing the server's messages.

We will take advantage of the sophisticated tokenizer and parser that are already present in TEX. In particular, we make use of *active characters*, that let TEX treat a single character as a macro. In the macro `\activecodes`, we set each possible message tag (`I`, `T`, ...) to be an active character and alias it to the macro we want to run upon receiving the message (the space after the `\def I` above is important for dealing with the space between the tag and the first argument). In the snippet above, we define `\init`, which takes eight tokens describing the size of the map, time limit, and information about sensor range and maximum speeds. Here we use TEX's delimited parameters to bind `#1...8` to the arguments so that we can save them for future use and then begin navigating.

```
\newlinechar`^^J
\def\send#1{%
  \message{^^JSEND: #1^^J}%% prints to stdout
}
\def\activecodes{
  \catcode`I=\active
  \def I {\init}%
  % ...
}
\def\init#1 #2 #3 #4 #5 #6 #7 #8 ;{%
  % ...
}
\def\main{%
  \message{^^JWANT^^J}%
  \begingroup              % make a local scope
    \activecodes           % activate characters for parsing
    \read 16 to \command   % stdin -> \command
    \command               % run \command
  \endgroup
  \main                    % loop
}
\main
```

Listing 1: Basic TEX routines for communicating with a network server via an appropriate Perl wrapper

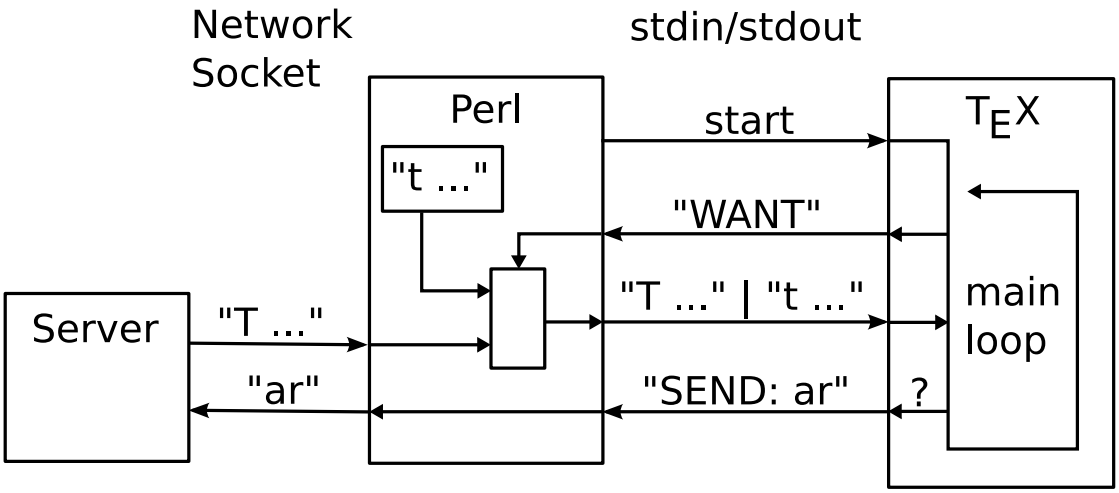

Figure 1: Connectivity diagram. The Perl wrapper opens a network socket and runs the TEX process. Each time TEX says "WANT", Perl sends back any recent network messages or else a pulse message ("`t` ⟨*time*⟩ `;`", generated interally) if none have arrived. Perl also forwards any messages from TEX back to the server.

The final macro, `\main`, stitches everything together. It is a tail-recursive loop that sends a message to Perl asking for data, activates the active character mechanism we use for parsing, waits for and reads input on stdin, and finally runs whatever is received before repeating the whole process. We can break out of this loop by redefining `\main`.

Since TEX's `\read`s are always blocking, it's important that we not waste any time waiting for input. Thus, in the event of network silence, we have Perl send back a "pulse" with just a timestamp so that TEX can get back to work navigating.

Pathfinding

At this point, we have reduced the problem to defining actions for each possible message. Since the pulse is the most frequently received message (as long as our efficiency isn't too terrible), we use it as the start of all the actions, leaving the others (initialization and telemetry data, in particular) to keep the variables up to date.

For each pulse, we use the velocity, direction, and turning rates to estimate the position and direction. Since acceleration rates aren't given, we ignore them. We have two modes, *running* and *swerving*, which determines what happens next. While running, we look ahead of the rover for anything we might hit. In the event of an imminent boulder or crater (we ignore the martians, since they appear to be

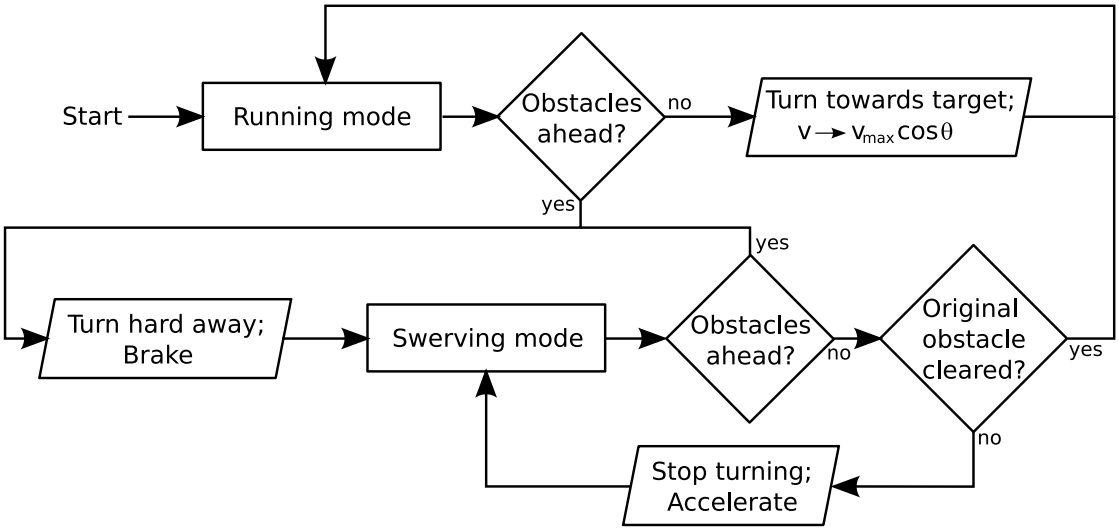

Figure 2: Flowchart of pathfinding logic.

rather dumb), we instantly switch into a self-preserving swerving mode. Otherwise, we turn towards the goal. We have two angular thresholds: if less than 1° off from the goal we go straight, from 1° to 10° we turn softly, and more than 10° we turn hard. We also scale the velocity by a cosine of the angle so that we slow down whenever we're off target.

In swerving more, we start by hitting the brakes and turning hard away from the obstacle. When the obstacle (plus a margin of saftey) is no longer in the way, we stop turning but remain traveling in a straight line until the obstacle is passed. Specifically, when the dot product between the velocity and the displacement to the object is negative. If at any point another obstruction occurs, we restart the swerving process.

Such a simple strategy is of course doomed to fail in any number of circumstances, but it seems to be robust as long as the obstacles are relatively sparse, as was the case in the example maps.

Development in TEX

Data types

Now that I've given the basic strategy, I'll talk a bit about the process of development in TEX, starting with a discussion of the available data types. We make use of five kinds of registers: conditionals store boolean values, counters store inte-

ger values, dimension registers store fixed-point decimal values up to 8192pt, and token registers and macros each store arbitrary lists of tokens (there are subtle differences in how each is accessed, but these are beyond the scope of this article).

TEX provides commands for basic arithmetic on counters. Dimensions can be multiplied and divided by counters, but there is no primitive way to multiply two dimension registers. Since dimensions are technically lengths, this would result in an area, which is meaningless to TEX. On the other hand, dimensions can be multiplied by fixed fractions, as in `1.2\baselineskip`. We can leverage this to multiply two dimensions by printing (expanding) the first dimension, removing the "pt" that comes along with it, and then attaching the second dimension. Listing 2 is adapted from a `comp.text.tex` post by Donald Arsineau [3].

```
\def\mul#1#2{\expandafter\removePT\the#1#2}
{\catcode`p=12 \catcode`t=12 \gdef\removePT#1pt{#1}}
```

Listing 2: A simple fixed-point multiplication routine.

The `\expandafter` sequences the operations such that the `\the#1` first becomes a number with units, e.g. `1.2pt`, and then `\removePT` gobbles up the `pt` so that the decimal number is left alone to multiply the dimension `#2`. Division is more complicated: we multiply the numerator and denominator by the largest power of two that doesn't cause an overflow, cast the denominator into a counter, and then divide.

The geometry of the problem requires also that we can extract square roots and evaluate sines, cosines, and arctangents, and this is another mess. I stole a square root routine, which basically uses Newton's method to solve $f(x) = x^2 - b = 0$, from a former colleague [4]. The trig functions are easily approximated with a quadratic over the interval $(0, 45°)$, and all the other values can be found with suitable transformations. See Listing 3 for the definition of `\sine`. This example demonstrates a bit of the flavor of TEX programming, as well as another complication so far unmentioned. The commands `\begingroup` and `\endgroup` set up a local scope so that we can use the dimension registers `\dimen@` and `\@tempdima` without fear of clobbering any data (alternately, we could allocate separate registers for each function, as long as they're never nested). The problem with this is that there's no good way to get the "return value" outside of this local scope, except by a global assignment, which is obviously undesirable. The solution again involves `\expandafter`: the temporary register (`\the\dimen@`) that holds the result is expanded in the expression `#1=\the\dimen@` *before* the `\endgroup` wipes out all our local assignments. Next the `\endgroup` happens, and then finally the assignment is carried out with the appropriate scope.

```
\catcode`\@=11 % allows @ to be used in macro names
% #1 is dimension register, in degrees; result overwrites input.
% Usage example: \dimen@=1.0pt  \sin\dimen@
\def\sine#1{
  \ifdim#1<0pt%          % sin(x) | x<0 = -sin(-x)
    #1=-#1\relax
    \sine#1\relax
    #1=-#1\relax
  \else
    \ifdim#1>45pt%       % sin(x) | x>45 = cos(90-x)
      \advance#1 by -90pt%
      \cosine#1%
    \else                % sin(x) | otherwise = x - .12*x^2
      \begingroup
        \dimen@=.017453#1\relax % convert to radians (pi/180)
        \@tempdima=\mul\dimen@\dimen@
        \advance\dimen@ by -.12\@tempdima
      \expandafter\endgroup
      \expandafter#1\expandafter=\the\dimen@\relax
    \fi
  \fi
}
```

Listing 3: The `\sine` function, defined in terms of the `\cosine` function, which is similar, differing only in the coefficients.

The last two types of data, token registers and macros, are also useful to us. A trivial usage is storing state (which direction we're turning; whether we're accelerating, braking, or coasting). A more interesting application is keeping track of the obstacles. In each telemetry report, the server sends information about all the objects in the rover's field of view. We store all this information in a token register in the form `\\{(`⟨*x*⟩`,`⟨*y*⟩`,`⟨*radius*⟩`)}` for each object, taking care not to store duplicates. We can then define `\\` to mean different things depending on what we want to do with this data, and then evaluate the token register. For instance, if we want to add a new object, we first define `\\` to check if the two objects are the same; or if we want to check for a possible collision course, we define `\\` to be the collision check macro [5]. This gives $O(n)$ access time for every operation, but more efficient solutions have other trade-offs. Listing 4 shows routines for adding a crater to our list, provided it's not already there.

The `\seeCrater` macro is rather straightforward, parsing the message from the server and changing it into our own format before calling `\append`. We see in `\append` the first use of a conditional variable, `\if@test`. Booleans in TEX actually consist of three macros: `\if`⟨*X*⟩ tests the condition, and `\`⟨*X*⟩`true` and `\`⟨*X*⟩`false` set the value. Another interesting feature is that our definition of `\\` is effectively a partial application of the `\eq` function.

Side effects

With all these difficulties, I would be remiss in not mentioning anything to TEX's advantage. Because TEX is designed for typesetting, one side effect of running it is that if we're not careful to avoid any extra spaces, we end up with a blank document as output. If we're even less careful, we might get text in the document that should have been parsed and executed (in my case, while working on the contest, this was actually decimal parts of dimensions that were truncated to integers without my realizing it).

On the other hand, once all the stray spaces are under control, there's nothing stopping us from using this side effect to produce meaningful output. While pictures are possible in plain TEX [6], they are not at all straightforward, so it makes sense to instead use LATEX's `picture` environment in which we can place dots at arbitrary points on the page. Any time we see a new object or update our position, we can issue the appropriate picture-drawing commands. The result can be seen in Figure 3.

```
\catcode`\@=11        % allow using @ in macro names
\newtoks\craters      % allocate a new token register

\def\seeCrater#1 #2 #3 {%    % message: "c <x> <y> <radius>"
  \dimen@=#3pt%              %   (as part of telemetry)
  \advance\dimen@ by 0.5pt%  % add radius of rover
  \append\craters{(#1,#2,\the\dimen@)}%
}

\def\eq#1#2{% if #1==#2 then set \if@test to true
  \begingroup
    \let\@result=\relax
    \def\@tempa{#1}
    \def\@tempb{#2}
    \ifx\@tempa\@tempb % test if two macros are equivalent
      \let\@result=\@testtrue
    \fi
  \expandafter\endgroup
  \@result
}

\def\append#1#2{% #1 is a token register, #2 is "(x,y,r)"
  \begingroup         % isolate scope
    \@testfalse       % any [] = False
    \def\\{\eq{#2}}%  % what to do with each element in #1
    \the#1%           % iterate over token register #1
    \if@test\else     % if #2 isn't yet in the list
      \global#1=\expandafter{%    then overwrite #1,
        \the#1\\{#2}% appending "\\{#2}"
      }%
    \fi
  \endgroup
}
```

Listing 4: Routines for adding a crater to a list of obstructions.

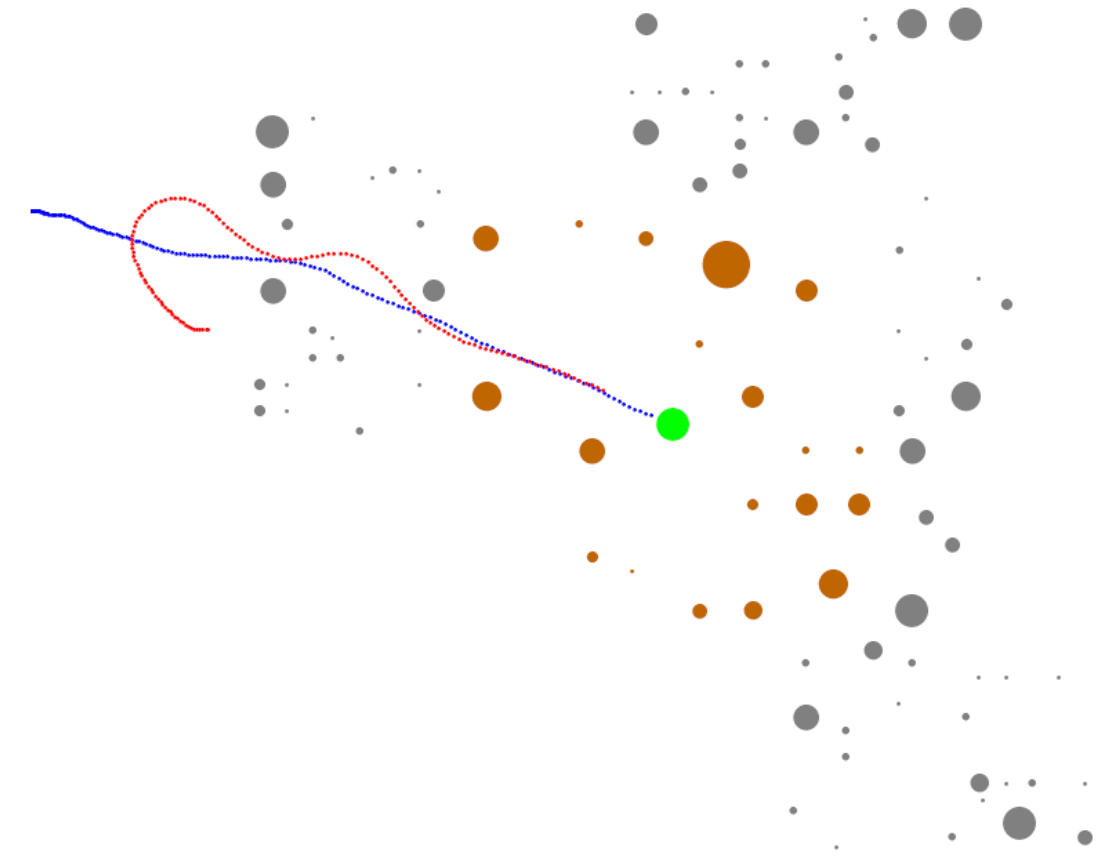

Figure 3: Typeset output of a mars rover trial run. The green circle in the center is the home base; brown circles are craters; gray circles are boulders; blue denotes the rover's path; and red denotes the martians (since we're ignoring them, they're just here for show).

Conclusions

I've discussed a few of the interesting features of writing a real-time application in TEX. Overall, I really can't recommend anyone else try it without significant masochistic tendencies (and then only for a limited time). On the other hand, it did provide a fun challenge for a weekend, and people's reactions were definitely worth it.

I learned a lot about TEX in writing this program. For one thing, I assumed TEX did not have any internal clock, so I used Perl to keep track of elapsed time. Afterwards, I was informed that pdfTEX has had a command `\pdfelapsedtime` for years. I alluded earlier to the fact that all operations on token registers are $O(n)$. This actually caused some efficiency problems in later trials because I saved the positions of all the obstacles, in case they would be useful. By the later trials, the list had grown rather large and TEX began to lag behind so much that it was no longer able to turn quickly enough to make it home or avoid obstacles. It would have been much better to forget about everything immediately after it left the field of view. Finally, in light of how ill-suited TEX is for math, I am convinced that METAFONT would have been a much better choice, but for the fact that I don't currently have the METAFONT skills to pull it off.

This is by no means the first instance of abusing TEX for something other than what it was intended for. Years ago, there was a simple BASIC interpreter written in TEX [7]. Just last fall, David Roundy and I wrote a crude build system in which TEX uses `\write18` to run shell commands and generate output from inline Python programs, but only if the listing was modified more recently than the output. Such exercises range from the absurd to the interesting, but generally there are much better ways to accomplish the same task. Nevertheless, as a physicist, I would like to end with a quote from Richard Feynman, who once said that "Every theoretical physicist who is any good knows six or seven theoretical representations for the same physics," [8] and I hope I'm not too far off base in applying this to programming as well.

About the author

Stephen Hicks is a senior Ph.D. student studying theoretical biophysics at Cornell University. He enjoys teaching, playing piano, kayaking, sailing, bridge, and lately hacking in TEX and Haskell.

References

[1] ICFP Programming Contest. `http://icfpcontest.org/`.

[2] Stephen Hicks. ICFP Contest 2008. http://sdh33b.blogspot.com/2008/07/icfp-contest-2008.html.

[3] Donald Arseneau. Fixed-point non-integer division (1993). http://groups.google.com/group/comp.text.tex/browse_thread/thread/95b55bc557d96b5. In comp.text.tex.

[4] Kevin Hamlen. Private correspondence.

[5] Donald Knuth. **The TEXbook**, chapter Appendix D, pages 378–379. Addison–Wesley (1984).

[6] Ibid, pages 389–390.

[7] Andrew Marc Greene. BASIX: An interpreter written in TEX. **TUGboat**, 11(3):pages 381–392 (1990). http://www.tug.org/TUGboat/Articles/tb11-3/contents11-3.html.

[8] Richard Feynman. **The Character of Physical Law**, chapter 7, page 168. MIT Press (1965).

The Typeclassopedia

by Brent Yorgey ⟨byorgey@cis.upenn.edu⟩

The standard Haskell libraries feature a number of type classes with algebraic or category-theoretic underpinnings. Becoming a fluent Haskell hacker requires intimate familiarity with them all, yet acquiring this familiarity often involves combing through a mountain of tutorials, blog posts, mailing list archives, and IRC logs.

The goal of this article is to serve as a starting point for the student of Haskell wishing to gain a firm grasp of its standard type classes. The essentials of each type class are introduced, with examples, commentary, and extensive references for further reading.

Introduction

Have you ever had any of the following thoughts?

- *What the heck is a monoid, and how is it different from a mon**a**d?*
- *I finally figured out how to use Parsec with do-notation, and someone told me I should use something called* `Applicative` *instead. Um, what?*
- *Someone in the #haskell IRC channel used* `(***)`*, and when I asked lambdabot to tell me its type, it printed out scary gobbledygook that didn't even fit on one line! Then someone used* `fmap fmap fmap` *and my brain exploded.*
- *When I asked how to do something I thought was really complicated, people started typing things like* `zip.ap fmap.(id &&& wtf)` *and the scary thing is that they worked! Anyway, I think those people must actually be robots because there's no way anyone could come up with that in two seconds off the top of their head.*

If you have, look no further! You, too, can write and understand concise, elegant, idiomatic Haskell code with the best of them.

There are two keys to an expert Haskell hacker's wisdom: 1. Understand the types. 2. Gain a deep intuition for each type class and its relationship to other type classes, backed up by familiarity with many examples.

It's impossible to overstate the importance of the first; the patient student of type signatures will uncover many profound secrets. Conversely, anyone ignorant of the types in their code is doomed to eternal uncertainty. "Hmm, it doesn't compile... maybe I'll stick in an `fmap` here... nope, let's see... maybe I need another `(.)` somewhere? ... um ..."

The second key—gaining deep intuition, backed by examples—is also important, but much more difficult to attain. A primary goal of this article is to set you on the road to gaining such intuition. However—

There is no royal road to Haskell.
—Euclid[1]

This article can only be a starting point, since good intuition comes from hard work, not from learning the right metaphor [1]. Anyone who reads and understands all of it will still have an arduous journey ahead—but sometimes a good starting point makes a big difference.

It should be noted that this is not a Haskell tutorial; it is assumed that the reader is already familiar with the basics of Haskell, including the standard `Prelude`, the type system, data types, and type classes.

Figure 1 on page 19 shows the type classes we will be discussing and their interrelationships. Solid arrows point from the general to the specific; that is, if there is an arrow from Foo to Bar it means that every Bar is (or should be, or can be made into) a Foo. Dotted arrows indicate some other sort of relationship. The solid double arrow indicates that `Monad` and `ArrowApply` are equivalent. `Pointed` and `Comonad` are greyed out since they are not actually (yet) in the standard Haskell libraries (they are in the category-extras library [2]).

One more note before we begin. I've seen "type class" written as one word, "typeclass," but let's settle this once and for all: the correct spelling uses two words (the title of this article notwithstanding), as evidenced by, for example, the Haskell 98 Revised Report [3], early papers on type classes [4, 5], and Hudak *et al.*'s history of Haskell [6].

We now begin with the simplest type class of all: `Functor`.

Functor

The `Functor` class [7] is the most basic and ubiquitous type class in the Haskell libraries. A simple intuition is that a `Functor` represents a "container" of some sort, along with the ability to apply a function uniformly to every element in the container. For example, a list is a container of elements, and we can apply a

[1] Well, he probably would have said it if he knew Haskell.

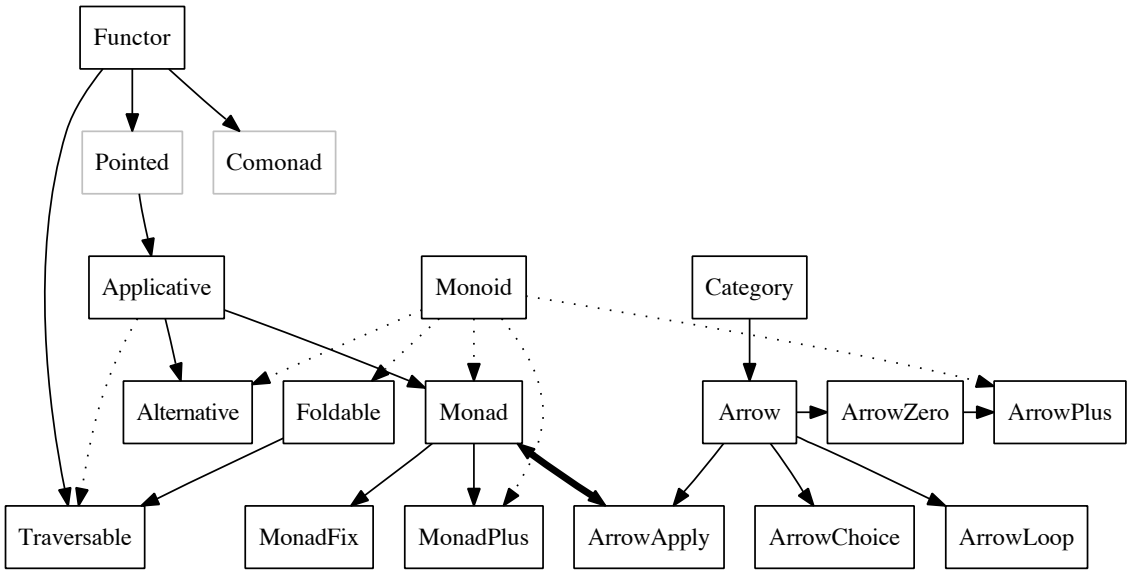

Figure 1: Relationships among standard Haskell type classes

function to every element of a list using `map`. A binary tree is also a container of elements, and it's not hard to come up with a way to recursively apply a function to every element in a tree.

Another intuition is that a `Functor` represents some sort of "computational context." This intuition is generally more useful, but is more difficult to explain, precisely because it is so general. Some examples later should help to clarify the `Functor`-as-context point of view.

In the end, however, a `Functor` is simply what it is defined to be; doubtless there are many examples of `Functor` instances that don't exactly fit either of the above intuitions. The wise student will focus their attention on definitions and examples, without leaning too heavily on any particular metaphor. Intuition will come, in time, on its own.

Definition

The type class declaration for `Functor` is shown in Listing 5. `Functor` is exported by the `Prelude`, so no special imports are needed to use it.

First, the `f a` and `f b` in the type signature for `fmap` tell us that `f` isn't just a type; it is a **type constructor** which takes another type as a parameter. (A more precise way to say this is that the **kind** of `f` must be `* -> *`.) For ex-

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

Listing 5: The `Functor` type class

ample, `Maybe` is such a type constructor: `Maybe` is not a type in and of itself, but requires another type as a parameter, like `Maybe Integer`. So it would not make sense to say `instance Functor Integer`, but it could make sense to say `instance Functor Maybe`.

Now look at the type of `fmap`: it takes any function from `a` to `b`, and a value of type `f a`, and outputs a value of type `f b`. From the container point of view, the intention is that `fmap` applies a function to each element of a container, without altering the structure of the container. From the context point of view, the intention is that `fmap` applies a function to a value without altering its context. Let's look at a few specific examples.

Instances

As noted before, the list constructor `[]` is a functor;[2] we can use the standard list function `map` to apply a function to each element of a list.[3] The `Maybe` type constructor is also a functor, representing a container which might hold a single element. The function `fmap g` has no effect on `Nothing` (there are no elements to which `g` can be applied), and simply applies `g` to the single element inside a `Just`. Alternatively, under the context interpretation, the list functor represents a context of nondeterministic choice; that is, a list can be thought of as representing a single value which is nondeterministically chosen from among several possibilities (the elements of the list). Likewise, the `Maybe` functor represents a context with possible failure. These instances are shown in Listing 6.

As an aside, in idiomatic Haskell code you will often see the letter `f` used to stand for both an arbitrary `Functor` and an arbitrary function. In this tutorial, I will use `f` only to represent `Functor`s, and `g` or `h` to represent functions, but you should be aware of the potential confusion. In practice, what `f` stands for should always be clear from the context, by noting whether it is part of a type or part of

[2] Recall that `[]` has two meanings in Haskell: it can either stand for the empty list, or, as here, it can represent the list type constructor (pronounced "list-of"). In other words, the type `[a]` (list-of-`a`) can also be written `([] a)`.

[3] You might ask why we need a separate `map` function. Why not just do away with the current list-only `map` function, and rename `fmap` to `map` instead? Well, that's a good question. The usual argument is that someone just learning Haskell, when using `map` incorrectly, would much rather see an error about lists than about `Functor`s.

```
instance Functor [] where
  fmap _ []     = []
  fmap g (x:xs) = g x : fmap g xs
  -- or we could just say  fmap = map

instance Functor Maybe where
  fmap _ Nothing  = Nothing
  fmap g (Just a) = Just (g a)
```

Listing 6: Two simple `Functor` instances

the code.

There are other `Functor` instances in the standard libraries; here are a few:[4]

- `Either e` is an instance of `Functor`; `Either e a` represents a container which can contain either a value of type `a`, or a value of type `e` (often representing some sort of error condition). It is similar to `Maybe` in that it represents possible failure, but it can carry some extra information about the failure as well.
- `((,) e)` represents a container which holds an "annotation" of type `e` along with the actual value it holds.
- `((->) e)`, the type of functions which take a value of type `e` as a parameter, is a `Functor`. It would be clearer to write it as `(e ->)`, by analogy with an operator section like `(1+)`, but that syntax is not allowed. However, you can certainly **think** of it as `(e ->)`. As a container, `(e -> a)` represents a (possibly infinite) set of values of `a`, indexed by values of `e`. Alternatively, and more usefully, `(e ->)` can be thought of as a context in which a value of type `e` is available to be consulted in a read-only fashion. This is also why `((->) e)` is sometimes referred to as the **reader monad**; more on this later.
- `IO` is a `Functor`; a value of type `IO a` represents a computation producing a value of type `a` which may have I/O effects. If `m` computes the value `x` while producing some I/O effects, then `fmap g m` will compute the value `g x` while producing the same I/O effects.
- Many standard types from the containers library [8] (such as `Tree`, `Map`, `Sequence`, and `Stream`) are instances of `Functor`. A notable exception is `Set`, which cannot be made a `Functor` in Haskell (although it is certainly a mathematical functor) since it requires an `Ord` constraint on its elements; `fmap` must be applicable to **any** types `a` and `b`.

[4] Note that some of these instances are not exported by the `Prelude`; to access them, you can import `Control.Monad.Instances`.

A good exercise is to implement `Functor` instances for `Either e`, `((,) e)`, and `((->) e)`.

Laws

As far as the Haskell language itself is concerned, the only requirement to be a `Functor` is an implementation of `fmap` with the proper type. Any sensible `Functor` instance, however, will also satisfy the **functor laws**, which are part of the definition of a mathematical functor. There are two, shown in Listing 7; together, these laws ensure that `fmap g` does not change the **structure** of a container, only the elements. Equivalently, and more simply, they ensure that `fmap g` changes a value without altering its context.[5]

```
fmap id = id
fmap (g . h) = fmap g . fmap h
```

Listing 7: The `Functor` laws

The first law says that mapping the identity function over every item in a container has no effect. The second says that mapping a composition of two functions over every item in a container is the same as first mapping one function, and then mapping the other.

As an example, the code shown in Listing 8 is a "valid" instance of `Functor` (it typechecks), but it violates the functor laws. Do you see why?

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

Listing 8: A lawless `Functor` instance

Any Haskeller worth their salt would reject the code in Listing 8 as a gruesome abomination.

Intuition

There are two fundamental ways to think about `fmap`. The first has already been touched on: it takes two parameters, a function and a container, and applies the

[5]Technically, these laws make `f` and `fmap` together an endofunctor on **Hask**, the category of Haskell types (ignoring $\perp$, which is a party pooper). [9]

function "inside" the container, producing a new container. Alternately, we can think of `fmap` as applying a function to a value in a context (without altering the context).

Just like all other Haskell functions of "more than one parameter," however, `fmap` is actually **curried**: it does not really take two parameters, but takes a single parameter and returns a function. For emphasis, we can write `fmap`'s type with extra parentheses: `fmap :: (a -> b) -> (f a -> f b)`. Written in this form, it is apparent that `fmap` transforms a "normal" function (`g :: a -> b`) into one which operates over containers/contexts (`fmap g :: f a -> f b`). This transformation is often referred to as a **lift**; `fmap` "lifts" a function from the "normal world" into the "`f` world."

Further reading

A good starting point for reading about the category theory behind the concept of a functor is the excellent Haskell wikibook page on category theory [9].

Pointed*

The `Pointed` type class represents **pointed functors**. It is not actually a type class in the standard libraries (hence the asterisk).[6] But it **could** be, and it's useful in understanding a few other type classes, notably `Applicative` and `Monad`, so let's pretend for a minute.

Given a `Functor`, the `Pointed` class represents the additional ability to put a value into a "default context." Often, this corresponds to creating a container with exactly one element, but it is more general than that. The type class declaration for `Pointed` is shown in Listing 9.

```
class Functor f => Pointed f where
  pure :: a -> f a     -- aka singleton, return, unit, point
```

Listing 9: The `Pointed` type class

Most of the standard `Functor` instances could also be instances of `Pointed`—for example, the `Maybe` instance of `Pointed` is `pure = Just`; there are many possible implementations for lists, the most natural of which is `pure x = [x]`; for `((->) e)` it is... well, I'll let you work it out. (Just follow the types!)

[6] It is, however, a type class in the category-extras library [2].

One example of a `Functor` which is not `Pointed` is `((,) e)`. If you try implementing `pure :: a -> (e,a)` you will quickly see why: since the type `e` is completely arbitrary, there is no way to generate a value of type `e` out of thin air! However, as we will see, `((,) e)` can be made `Pointed` if we place an additional restriction on `e` which allows us to generate a default value of type `e` (the most common solution is to make `e` an instance of `Monoid`).

The `Pointed` class has only one law, shown in Listing 10.[7]

```
fmap g . pure  =  pure . g
```

Listing 10: The `Pointed` law

However, you need not worry about it: this law is actually a so-called "free theorem" guaranteed by parametricity [10]; it's impossible to write an instance of `Pointed` which does not satisfy it.[8]

Applicative

A somewhat newer addition to the pantheon of standard Haskell type classes, applicative functors [11] represent an abstraction lying exactly in between `Functor` and `Monad`, first described by McBride and Paterson [12]. The title of McBride and Paterson's classic paper, **Applicative Programming with Effects**, gives a hint at the intended intuition behind the `Applicative` type class. It encapsulates certain sorts of "effectful" computations in a functionally pure way, and encourages an "applicative" programming style. Exactly what these things mean will be seen later.

Definition

The `Applicative` class adds a single capability to `Pointed` functors. Recall that `Functor` allows us to lift a "normal" function to a function on computational contexts. But `fmap` doesn't allow us to apply a function which is itself in a context to a value in another context. `Applicative` gives us just such a tool. Listing 11 shows the type class declaration for `Applicative`, which is defined in `Control.Applicative`. Note that every `Applicative` must also be a `Functor`. In fact, as we will see, `fmap` can be implemented using the `Applicative` methods, so

[7] For those interested in category theory, this law states precisely that `pure` is a natural transformation from the identity functor to `f`.

[8] ... modulo $\perp$, `seq`, and assuming a lawful `Functor` instance.

every `Applicative` is a functor whether we like it or not; the `Functor` constraint forces us to be honest.

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

Listing 11: The `Applicative` type class

As always, it's crucial to understand the type signature of `(<*>)`. The best way of thinking about it comes from noting that the type of `(<*>)` is similar to the type of `($)`,[9] but with everything enclosed in an `f`. In other words, `(<*>)` is just function application within a computational context. The type of `(<*>)` is also very similar to the type of `fmap`; the only difference is that the first parameter is `f (a -> b)`, a function in a context, instead of a "normal" function `(a -> b)`.

Of course, `pure` looks rather familiar. If we actually had a `Pointed` type class, `Applicative` could instead be defined as shown in Listing 12.

```
class Pointed f => Applicative' f where
  (<*>) :: f (a -> b) -> f a -> f b
```

Listing 12: Alternate definition of `Applicative` using `Pointed`

Laws

There are several laws that `Applicative` instances should satisfy [11, 12], but only one is crucial to developing intuition, because it specifies how `Applicative` should relate to `Functor` (the other four mostly specify the exact sense in which `pure` deserves its name). This law is shown in Listing 13.

```
fmap g x = pure g <*> x
```

Listing 13: Law relating `Applicative` to `Functor`

The law says that mapping a pure function `g` over a context `x` is the same as first injecting `g` into a context with `pure`, and then applying it to `x` with `(<*>)`. In other

[9] Recall that `($)` is just function application: `f $ x = f x`.

words, we can decompose `fmap` into two more atomic operations: injection into a context, and application within a context. The `Control.Applicative` module also defines `(<$>)` as a synonym for `fmap`, so the above law can also be expressed as `g <$> x = pure g <*> x`.

Instances

Most of the standard types which are instances of `Functor` are also instances of `Applicative`.

`Maybe` can easily be made an instance of `Applicative`; writing such an instance is left as an exercise for the reader.

The list type constructor `[]` can actually be made an instance of `Applicative` in two ways; essentially, it comes down to whether we want to think of lists as ordered collections of elements, or as contexts representing multiple results of a nondeterministic computation [13].

Let's first consider the collection point of view. Since there can only be one instance of a given type class for any particular type, one or both of the list instances of `Applicative` need to be defined for a `newtype` wrapper; as it happens, the nondeterministic computation instance is the default, and the collection instance is defined in terms of a `newtype` called `ZipList`. This instance is shown in Listing 14.

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
  pure = undefined   -- exercise
  (ZipList gs) <*> (ZipList xs) = ZipList (zipWith ($) gs xs)
```

Listing 14: `ZipList` instance of `Applicative`

To apply a list of functions to a list of inputs with `(<*>)`, we just match up the functions and inputs elementwise, and produce a list of the resulting outputs. In other words, we "zip" the lists together with function application, `($)`; hence the name `ZipList`. As an exercise, determine the correct definition of `pure`—there is only one implementation that satisfies the law in Listing 13.

The other `Applicative` instance for lists, based on the nondeterministic computation point of view, is shown in Listing 15. Instead of applying functions to inputs pairwise, we apply each function to all the inputs in turn, and collect all the results in a list.

```
instance Applicative [] where
  pure x = [x]
  gs <*> xs = [ g x | g <- gs, x <- xs ]
```

Listing 15: `[]` instance of `Applicative`

Now we can write nondeterministic computations in a natural style. To add the numbers `3` and `4` deterministically, we can of course write `(+) 3 4`. But suppose instead of `3` we have a nondeterministic computation that might result in `2`, `3`, or `4`; then we can write

```
pure (+) <*> [2,3,4] <*> pure 4
```

or, more idiomatically,

```
(+) <$> [2,3,4] <*> pure 4.
```

There are several other `Applicative` instances as well:

- `IO` is an instance of `Applicative`, and behaves exactly as you would think: when `g <$> m1 <*> m2 <*> m3` is executed, the effects from the `mi`'s happen in order from left to right.
- `((,) a)` is an `Applicative`, as long as `a` is an instance of `Monoid` (page 39). The `a` values are accumulated in parallel with the computation.
- The `Applicative` module defines the `Const` type constructor; a value of type `Const a b` simply contains an `a`. This is an instance of `Applicative` for any `Monoid a`; this instance becomes especially useful in conjunction with things like `Foldable` (page 44).
- The `WrappedMonad` and `WrappedArrow` newtypes make any instances of `Monad` (page 29) or `Arrow` (page 51) respectively into instances of `Applicative`; as we will see when we study those type classes, both are strictly more expressive than `Applicative`, in the sense that the `Applicative` methods can be implemented in terms of their methods.

Intuition

McBride and Paterson's paper introduces the notation $[\![g\ x_1\ x_2\ \cdots\ x_n]\!]$ to denote function application in a computational context. If each x_i has type $f\ t_i$ for some applicative functor f, and g has type $t_1 \to t_2 \to \cdots \to t_n \to t$, then the entire expression $[\![g\ x_1\ \cdots\ x_n]\!]$ has type $f\ t$. You can think of this as applying a function to multiple "effectful" arguments. In this sense, the double bracket notation is a

generalization of `fmap`, which allows us to apply a function to a single argument in a context.

Why do we need `Applicative` to implement this generalization of `fmap`? Suppose we use `fmap` to apply `g` to the first parameter `x1`. Then we get something of type `f (t2 -> ... t)`, but now we are stuck: we can't apply this function-in-a-context to the next argument with `fmap`. However, this is precisely what `(<*>)` allows us to do.

This suggests the proper translation of the idealized notation $[\![g\ x_1\ x_2\ \cdots\ x_n]\!]$ into Haskell, namely

```
g <$> x1 <*> x2 <*> ... <*> xn,
```

recalling that `Control.Applicative` defines `(<$>)` as a convenient infix shorthand for `fmap`. This is what is meant by an "applicative style"—effectful computations can still be described in terms of function application; the only difference is that we have to use the special operator `(<*>)` for application instead of simple juxtaposition.

Further reading

There are many other useful combinators in the standard libraries implemented in terms of `pure` and `(<*>)`: for example, `(*>)`, `(<*)`, `(<**>)`, `(<$)`, and so on [11]. Judicious use of such secondary combinators can often make code using `Applicative`s much easier to read.

McBride and Paterson's original paper [12] is a treasure-trove of information and examples, as well as some perspectives on the connection between `Applicative` and category theory. Beginners will find it difficult to make it through the entire paper, but it is extremely well-motivated—even beginners will be able to glean something from reading as far as they are able.

Conal Elliott has been one of the biggest proponents of `Applicative`. For example, the Pan library for functional images [14] and the reactive library for functional reactive programming (FRP) [15] make key use of it; his blog also contains many examples of `Applicative` in action [16]. Building on the work of McBride and Paterson, Elliott also built the TypeCompose library [17], which embodies the observation (among others) that `Applicative` types are closed under composition; therefore, `Applicative` instances can often be automatically derived for complex types built out of simpler ones.

Although the Parsec parsing library [18, 19] was originally designed for use as a monad, in its most common use cases an `Applicative` instance can be used to great effect; Bryan O'Sullivan's blog post is a good starting point [20]. If the extra power provided by `Monad` isn't needed, it's usually a good idea to use `Applicative` instead.

A couple other nice examples of `Applicative` in action include the ConfigFile and HSQL libraries [21] and the formlets library [22].

Monad

It's a safe bet that if you're reading this article, you've heard of monads—although it's quite possible you've never heard of `Applicative` before, or `Arrow`, or even `Monoid`. Why are monads such a big deal in Haskell? There are several reasons.

- Haskell does, in fact, single out monads for special attention by making them the framework in which to construct I/O operations.
- Haskell also singles out monads for special attention by providing a special syntactic sugar for monadic expressions: the `do`-notation.
- `Monad` has been around longer than various other abstract models of computation such as `Applicative` or `Arrow`.
- The more monad tutorials there are, the harder people think monads must be, and the more new monad tutorials are written by people who think they finally "get" monads [1].

I will let you judge for yourself whether these are good reasons.

In the end, despite all the hoopla, `Monad` is just another type class. Let's take a look at its definition.

Definition

The type class declaration for `Monad` [23] is shown in Listing 16. The `Monad` type class is exported by the `Prelude`, along with a few standard instances. However, many utility functions are found in `Control.Monad`, and there are also several instances (such as `((->) e)`) defined in `Control.Monad.Instances`.

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

  fail   :: String -> m a
```

Listing 16: The `Monad` type class

Let's examine the methods in the `Monad` class one by one. The type of `return` should look familiar; it's the same as `pure`. Indeed, `return` **is** `pure`, but with

an unfortunate name. (Unfortunate, since someone coming from an imperative programming background might think that `return` is like the C or Java keyword of the same name, when in fact the similarities are minimal.) From a mathematical point of view, every monad is a pointed functor (indeed, an applicative functor), but for historical reasons, the `Monad` type class declaration unfortunately does not require this.

We can see that `(>>)` is a specialized version of `(>>=)`, with a default implementation given. It is only included in the type class declaration so that specific instances of `Monad` can override the default implementation of `(>>)` with a more efficient one, if desired. Also, note that although `_ >> n = n` would be a type-correct implementation of `(>>)`, it would not correspond to the intended semantics: the intention is that `m >> n` ignores the **result** of `m`, but not its **effects**.

The `fail` function is an awful hack that has no place in the `Monad` class; more on this later.

The only really interesting thing to look at—and what makes `Monad` strictly more powerful than `Pointed` or `Applicative`—is `(>>=)`, which is often called **bind**. An alternative definition of `Monad` could look like Listing 17.

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

Listing 17: An alternative definition of `Monad`

We could spend a while talking about the intuition behind `(>>=)`—and we will. But first, let's look at some examples.

Instances

Even if you don't understand the intuition behind the `Monad` class, you can still create instances of it by just seeing where the types lead you. You may be surprised to find that this actually gets you a long way towards understanding the intuition; at the very least, it will give you some concrete examples to play with as you read more about the `Monad` class in general. The first few examples are from the standard `Prelude`; the remaining examples are from the monad transformer library (mtl) [24].

- The simplest possible instance of `Monad` is `Identity` [25], which is described in Dan Piponi's highly recommended blog post on "The Trivial Monad" [26]. Despite being "trivial," it is a great introduction to the `Monad` type class, and contains some good exercises to get your brain working.

- The next simplest instance of `Monad` is `Maybe`. We already know how to write `return`/`pure` for `Maybe`. So how do we write `(>>=)`? Well, let's think about its type. Specializing for `Maybe`, we have
  `(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b.`
  If the first argument to `(>>=)` is `Just x`, then we have something of type `a` (namely, `x`), to which we can apply the second argument—resulting in a `Maybe b`, which is exactly what we wanted. What if the first argument to `(>>=)` is `Nothing`? In that case, we don't have anything to which we can apply the `a -> Maybe b` function, so there's only one thing we can do: yield `Nothing`. This instance is shown in Listing 18. We can already get a bit of intuition as to what is going on here: if we build up a computation by chaining together a bunch of functions with `(>>=)`, as soon as any one of them fails, the entire computation will fail (because `Nothing >>= f` is `Nothing`, no matter what `f` is). The entire computation succeeds only if all the constituent functions individually succeed. So the `Maybe` monad models computations which may fail.

```
instance Monad Maybe where
  return = Just
  (Just x) >>= g = g x
  Nothing  >>= _ = Nothing
```

Listing 18: The `Maybe` instance of `Monad`

- The `Monad` instance for the list constructor `[]` is similar to its `Applicative` instance; I leave its implementation as an exercise. Follow the types!
- Of course, the `IO` constructor is famously a `Monad`, but its implementation is somewhat magical, and may in fact differ from compiler to compiler. It is worth emphasizing that the `IO` monad is the **only** monad which is magical. It allows us to build up, in an entirely pure way, values representing possibly effectful computations. The special value `main`, of type `IO ()`, is taken by the runtime and actually executed, producing actual effects. Every other monad is functionally pure, and requires no special compiler support. We often speak of monadic values as "effectful computations," but this is because some monads allow us to write code **as if** it has side effects, when in fact the monad is hiding the plumbing which allows these apparent side effects to be implemented in a functionally pure way.
- As mentioned earlier, `((->) e)` is known as the **reader monad**, since it describes computations in which a value of type `e` is available as a read-only environment. It is worth trying to write a `Monad` instance for `((->) e)`

yourself.

The `Control.Monad.Reader` module [27] provides the `Reader e a` type, which is just a convenient `newtype` wrapper around `(e -> a)`, along with an appropriate `Monad` instance and some `Reader`-specific utility functions such as `ask` (retrieve the environment), `asks` (retrieve a function of the environment), and `local` (run a subcomputation under a different environment).

- The `Control.Monad.Writer` module [28] provides the `Writer` monad, which allows information to be collected as a computation progresses. `Writer w a` is isomorphic to `(a,w)`, where the output value `a` is carried along with an annotation or "log" of type `w`, which must be an instance of `Monoid` (page 39); the special function `tell` performs logging.
- The `Control.Monad.State` module [29] provides the `State s a` type, a `newtype` wrapper around `s -> (a,s)`. Something of type `State s a` represents a stateful computation which produces an `a` but can access and modify the state of type `s` along the way. The module also provides `State`-specific utility functions such as `get` (read the current state), `gets` (read a function of the current state), `put` (overwrite the state), and `modify` (apply a function to the state).
- The `Control.Monad.Cont` module [30] provides the `Cont` monad, which represents computations in continuation-passing style. It can be used to suspend and resume computations, and to implement non-local transfers of control, co-routines, other complex control structures—all in a functionally pure way. `Cont` has been called the "mother of all monads" [31] because of its universal properties.

Intuition

Let's look more closely at the type of `(>>=)`. The basic intuition is that it combines two computations into one larger computation. The first argument, `m a`, is the first computation. However, it would be boring if the second argument were just an `m b`; then there would be no way for the computations to interact with one another. So, the second argument to `(>>=)` has type `a -> m b`: a function of this type, given a **result** of the first computation, can produce a second computation to be run. In other words, `x >>= k` is a computation which runs `x`, and then uses the result(s) of `x` to **decide** what computation to run second, using the output of the second computation as the result of the entire computation.

Intuitively, it is this ability to use the output from previous computations to decide what computations to run next that makes `Monad` more powerful than `Applicative`. The structure of an `Applicative` computation is fixed, whereas the structure of a `Monad` computation can change based on intermediate results.

To see the increased power of `Monad` from a different point of view, let's see what

happens if we try to implement `(>>=)` in terms of `fmap`, `pure`, and `(<*>)`. We are given a value `x` of type `m a`, and a function `k` of type `a -> m b`, so the only thing we can do is apply `k` to `x`. We can't apply it directly, of course; we have to use `fmap` to lift it over the `m`. But what is the type of `fmap k`? Well, it's `m a -> m (m b)`. So after we apply it to `x`, we are left with something of type `m (m b)`—but now we are stuck; what we really want is an `m b`, but there's no way to get there from here. We can **add** `m`'s using `pure`, but we have no way to **collapse** multiple `m`'s into one.

This ability to collapse multiple `m`'s is exactly the ability provided by the function `join :: m (m a) -> m a`, and it should come as no surprise that an alternative definition of `Monad` can be given in terms of `join`, as shown in Listing 19.

```
class Applicative m => Monad'' m where
  join :: m (m a) -> m a
```

Listing 19: An alternative definition of `Monad` in terms of `join`

In fact, monads in category theory are defined in terms of `return`, `fmap`, and `join` (often called η, T, and μ in the mathematical literature). Haskell uses the equivalent formulation in terms of `(>>=)` instead of `join` since it is more convenient to use; however, sometimes it can be easier to think about `Monad` instances in terms of `join`, since it is a more "atomic" operation. (For example, `join` for the list monad is just `concat`.) An excellent exercise is to implement `(>>=)` in terms of `fmap` and `join`, and to implement `join` in terms of `(>>=)`.

Utility functions

The `Control.Monad` module [32] provides a large number of convenient utility functions, all of which can be implemented in terms of the basic `Monad` operations (`return` and `(>>=)` in particular). We have already seen one of them, namely, `join`. We also mention some other noteworthy ones here; implementing these utility functions oneself is a good exercise. For a more detailed guide to these functions, with commentary and example code, see Henk-Jan van Tuyl's tour [33].

- `liftM :: Monad m => (a -> b) -> m a -> m b`. This should be familiar; of course, it is just `fmap`. The fact that we have both `fmap` and `liftM` is an unfortunate consequence of the fact that the `Monad` type class does not require a `Functor` instance, even though mathematically speaking, every monad is a functor. However, `fmap` and `liftM` are essentially interchangeable, since it is a bug (in a social rather than technical sense) for any type to be an instance of `Monad` without also being an instance of `Functor`.

- `ap :: Monad m => m (a -> b) -> m a -> m b` should also be familiar: it is equivalent to `(<*>)`, justifying the claim that the `Monad` interface is strictly more powerful than `Applicative`. We can make any `Monad` into an instance of `Applicative` by setting `pure = return` and `(<*>) = ap`.
- `sequence :: Monad m => [m a] -> m [a]` takes a list of computations and combines them into one computation which collects a list of their results. It is again something of a historical accident that `sequence` has a `Monad` constraint, since it can actually be implemented only in terms of `Applicative`. There is also an additional generalization of `sequence` to structures other than lists, which will be discussed in the section on `Traversable` (page 47).
- `replicateM :: Monad m => Int -> m a -> m [a]` is simply a combination of `replicate` and `sequence`.
- `when :: Monad m => Bool -> m () -> m ()` conditionally executes a computation, evaluating to its second argument if the test is `True`, and to `return ()` if the test is `False`. A collection of other sorts of monadic conditionals can be found in the IfElse package [34].
- `mapM :: Monad m => (a -> m b) -> [a] -> m [b]` maps its first argument over the second, and `sequence`s the results. The `forM` function is just `mapM` with its arguments reversed; it is called `forM` since it models generalized `for` loops: the list `[a]` provides the loop indices, and the function `a -> m b` specifies the "body" of the loop for each index.
- `(=<<) :: Monad m => (a -> m b) -> m a -> m b` is just `(>>=)` with its arguments reversed; sometimes this direction is more convenient since it corresponds more closely to function application.
- `(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c` is sort of like function composition, but with an extra `m` on the result type of each function, and the arguments swapped. We'll have more to say about this operation later.
- The `guard` function is for use with instances of `MonadPlus`, which is discussed at the end of the `Monoid` section.

Many of these functions also have "underscored" variants, such as `sequence_` and `mapM_`; these variants throw away the results of the computations passed to them as arguments, using them only for their side effects.

Laws

There are several laws that instances of `Monad` should satisfy [35]. The standard presentation is shown in Listing 20.

The first and second laws express the fact that `return` behaves nicely: if we inject a value `a` into a monadic context with `return`, and then bind to `k`, it is the same as just applying `k` to `a` in the first place; if we bind a computation `m` to

```
return a >>= k  =  k a
m >>= return    =  m
m >>= (\x -> k x >>= h)  =  (m >>= k) >>= h

fmap f xs  =  xs >>= return . f  =  liftM f xs
```

Listing 20: The `Monad` laws

`return`, nothing changes. The third law essentially says that `(>>=)` is associative, sort of. The last law ensures that `fmap` and `liftM` are the same for types which are instances of both `Functor` and `Monad`—which, as already noted, should be every instance of `Monad`.

However, the presentation of the above laws, especially the third, is marred by the asymmetry of `(>>=)`. It's hard to look at the laws and see what they're really saying. I prefer a much more elegant version of the laws, which is formulated in terms of `(>=>)`.[10] Recall that `(>=>)` "composes" two functions of type `a -> m b` and `b -> m c`. You can think of something of type `a -> m b` (roughly) as a function from `a` to `b` which may also have some sort of effect in the context corresponding to `m`. (Note that `return` is such a function.) `(>=>)` lets us compose these "effectful functions," and we would like to know what properties `(>=>)` has. The monad laws reformulated in terms of `(>=>)` are shown in Listing 21.

```
return >=> g  =  g
g >=> return  =  g
(g >=> h) >=> k  =  g >=> (h >=> k)
```

Listing 21: The `Monad` laws, reformulated in terms of `(>=>)`

Ah, much better! The laws simply state that `return` is the identity of `(>=>)`, and that `(>=>)` is associative.[11] Working out the equivalence between these two formulations, given the definition `g >=> h = \x -> g x >>= h`, is left as an exercise.

There is also a formulation of the monad laws in terms of `fmap`, `return`, and `join`; for a discussion of this formulation, see the Haskell wikibook page on category theory [9].

[10] I like to pronounce this operator "fish," but that's probably not the canonical pronunciation...

[11] As fans of category theory will note, these laws say precisely that functions of type `a -> m b` are the arrows of a category with `(>=>)` as composition! Indeed, this is known as the **Kleisli category** of the monad `m`. It will come up again when we discuss `Arrow`s.

`do` notation

Haskell's special `do` notation supports an "imperative style" of programming by providing syntactic sugar for chains of monadic expressions. The genesis of the notation lies in realizing that something like `a >>= \x -> b >> c >>= \y -> d` can be more readably written by putting successive computations on separate lines:

```
a >>= \x ->
b >>
c >>= \y ->
d
```

This emphasizes that the overall computation consists of four computations `a`, `b`, `c`, and `d`, and that `x` is bound to the result of `a`, and `y` is bound to the result of `c` (`b`, `c`, and `d` are allowed to refer to `x`, and `d` is allowed to refer to `y` as well). From here it is not hard to imagine a nicer notation:

```
do { x <- a ;
     b      ;
     y <- c ;
     d
   }
```

(The curly braces and semicolons may optionally be omitted; the Haskell parser uses layout to determine where they should be inserted.) This discussion should make clear that `do` notation is just syntactic sugar. In fact, `do` blocks are recursively translated into monad operations (almost) as shown in Listing 22.

$$\begin{aligned}
\texttt{do } e &\longrightarrow e \\
\texttt{do } \{e; \mathit{stmts}\} &\longrightarrow e \texttt{ >> do } \{\mathit{stmts}\} \\
\texttt{do } \{v \texttt{ <- } e; \mathit{stmts}\} &\longrightarrow e \texttt{ >>= \}v \texttt{ -> do } \{\mathit{stmts}\} \\
\texttt{do } \{\texttt{let } \mathit{decls}; \mathit{stmts}\} &\longrightarrow \texttt{let } \mathit{decls} \texttt{ in do } \{\mathit{stmts}\}
\end{aligned}$$

Listing 22: Desugaring of `do` blocks (almost)

This is not quite the whole story, since v might be a pattern instead of a variable. For example, one can write

```
do (x:xs) <- foo
   bar x
```

but what happens if `foo` produces an empty list? Well, remember that ugly `fail` function in the `Monad` type class declaration? That's what happens. See section 3.14 of the Haskell Report for the full details [3]. See also the discussion of `MonadPlus` and `MonadZero` (page 42).

A final note on intuition: `do` notation plays very strongly to the "computational context" point of view rather than the "container" point of view, since the binding notation `x <- m` is suggestive of "extracting" a single `x` from `m` and doing something with it. But `m` may represent some sort of a container, such as a list or a tree; the meaning of `x <- m` is entirely dependent on the implementation of `(>>=)`. For example, if `m` is a list, `x <- m` actually means that `x` will take on each value from the list in turn.

Monad transformers

One would often like to be able to combine two monads into one: for example, to have stateful, nondeterministic computations (`State + []`), or computations which may fail and can consult a read-only environment (`Maybe + Reader`), and so on. Unfortunately, monads do not compose as nicely as applicative functors (yet another reason to use `Applicative` if you don't need the full power that `Monad` provides), but some monads can be combined in certain ways.

The monad transformer library [24] provides a number of **monad transformers**, such as `StateT`, `ReaderT`, `ErrorT` [36], and (soon) `MaybeT`, which can be applied to other monads to produce a new monad with the effects of both. For example, `StateT s Maybe` is an instance of `Monad`; computations of type `StateT s Maybe a` may fail, and have access to a mutable state of type `s`. These transformers can be multiply stacked. One thing to keep in mind while using monad transformers is that the order of composition matters. For example, when a `StateT s Maybe a` computation fails, the state ceases being updated; on the other hand, the state of a `MaybeT (State s) a` computation may continue to be modified even after the computation has failed. (This may seem backwards, but it is correct. Monad transformers build composite monads "inside out"; for example, `MaybeT (State s) a` is isomorphic to `s -> Maybe (a, s)`. Lambdabot has an indispensable `@unmtl` command which you can use to "unpack" a monad transformer stack in this way.)

All monad transformers should implement the `MonadTrans` type class (Listing 23), defined in `Control.Monad.Trans`. It allows arbitrary computations in the base monad `m` to be "lifted" into computations in the transformed monad `t m`. (Note that type application associates to the left, just like function application, so `t m a = (t m) a`. As an exercise, you may wish to work out `t`'s kind, which is rather more interesting than most of the kinds we've seen up to this point.) However, you should only have to think about `MonadTrans` when defining your own monad transformers, not when using predefined ones.

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

Listing 23: The `MonadTrans` type class

There are also type classes such as `MonadState`, which provides state-specific methods like `get` and `put`, allowing you to conveniently use these methods not only with `State`, but with any monad which is an instance of `MonadState`—including `MaybeT (State s)`, `StateT s (ReaderT r IO)`, and so on. Similar type classes exist for `Reader`, `Writer`, `Cont`, `IO`, and others.[12]

There are two excellent references on monad transformers. Martin Grabmüller's **Monad Transformers Step by Step** [37] is a thorough description, with running examples, of how to use monad transformers to elegantly build up computations with various effects. Cale Gibbard's article on how to use monad transformers [38] is more practical, describing how to structure code using monad transformers to make writing it as painless as possible. Another good starting place for learning about monad transformers is a blog post by Dan Piponi [39].

MonadFix

The `MonadFix` class describes monads which support the special fixpoint operation `mfix :: (a -> m a) -> m a`, which allows the output of monadic computations to be defined via recursion. This is supported in GHC and Hugs by a special "recursive do" notation, `mdo`. For more information, see Levent Erkök's thesis, **Value Recursion in Monadic Computations** [40].

Further reading

Philip Wadler was the first to propose using monads to structure functional programs [41]. His paper is still a readable introduction to the subject.

Much of the monad transformer library (mtl) [24], including the `Reader`, `Writer`, `State`, and other monads, as well as the monad transformer framework itself, was inspired by Mark Jones's classic paper **Functional Programming with Overloading and Higher-Order Polymorphism** [42]. It's still very much worth a read—and highly readable—after almost fifteen years.

[12] The only problem with this scheme is the quadratic number of instances required as the number of standard monad transformers grows—but as the current set of standard monad transformers seems adequate for most common use cases, this may not be that big of a deal.

There are, of course, numerous monad tutorials of varying quality [43, 44, 45, 46, 47, 48, 49, 50, 51, 52]. A few of the best include Cale Gibbard's **Monads as containers** [44] and **Monads as computation** [51]; Jeff Newbern's **All About Monads** [43], a comprehensive guide with lots of examples; and Dan Piponi's **You could have invented monads!**, which features great exercises [47]. If you just want to know how to use `IO`, you could consult the **Introduction to IO** [53]. Even this is just a sampling; a more complete list can be found on the Haskell wiki [54]. (All these monad tutorials have prompted some parodies [55] as well as other kinds of backlash [56, 1].) Other good monad references which are not necessarily tutorials include Henk-Jan van Tuyl's tour of the functions in `Control.Monad` [33], Dan Piponi's "field guide" [57], and Tim Newsham's **What's a Monad?** [58]. There are also many blog articles which have been written on various aspects of monads; a collection of links can be found on the Haskell wiki [59].

One of the quirks of the `Monad` class and the Haskell type system is that it is not possible to straightforwardly declare `Monad` instances for types which require a class constraint on their data, even if they are monads from a mathematical point of view. For example, `Data.Set` requires an `Ord` constraint on its data, so it cannot be easily made an instance of `Monad`. A solution to this problem was first described by Eric Kidd [60], and later made into a library by Ganesh Sittampalam and Peter Gavin [61].

There are many good reasons for eschewing `do` notation; some have gone so far as to consider it harmful [62].

Monads can be generalized in various ways; for an exposition of one possibility, **parameterized monads**, see Robert Atkey's paper on the subject [63], or Dan Piponi's exposition [64].

For the categorically inclined, monads can be viewed as monoids [65] and also as closure operators [66]. Derek Elkins's article in this issue of the Monad.Reader [67] contains an exposition of the category-theoretic underpinnings of some of the standard `Monad` instances, such as `State` and `Cont`. There is also an alternative way to compose monads, using coproducts, as described by Lüth and Ghani [68], although this method has not (yet?) seen widespread use.

Links to many more research papers related to monads can be found on the Haskell wiki [69].

Monoid

A monoid is a set S together with a binary operation $\oplus$ which combines elements from S. The $\oplus$ operator is required to be associative (that is, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$, for any a, b, c which are elements of S), and there must be some element of S which is the identity with respect to $\oplus$. (If you are familiar with group theory, a monoid is

like a group without the requirement that inverses exist.) For example, the natural numbers under addition form a monoid: the sum of any two natural numbers is a natural number; $(a + b) + c = a + (b + c)$ for any natural numbers a, b, and c; and zero is the additive identity. The integers under multiplication also form a monoid, as do natural numbers under max, Boolean values under conjunction and disjunction, lists under concatenation, functions from a set to itself under composition.... Monoids show up all over the place, once you know to look for them.

Definition

The definition of the `Monoid` type class (defined in `Data.Monoid`) [70] is shown in Listing 24.

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Listing 24: The `Monoid` type class

The `mempty` value specifies the identity element of the monoid, and `mappend` is the binary operation. The default definition for `mconcat` "reduces" a list of elements by combining them all with `mappend`, using a right fold. It is only in the `Monoid` class so that specific instances have the option of providing an alternative, more efficient implementation; usually, you can safely ignore `mconcat` when creating a `Monoid` instance, since its default definition will work just fine.

The `Monoid` methods are rather unfortunately named; they are inspired by the list instance of `Monoid`, where indeed `mempty = []` and `mappend = (++)`, but this is misleading since many monoids have little to do with appending [71].

Laws

Of course, every `Monoid` instance should actually be a monoid in the mathematical sense, which implies the laws shown in Listing 25.

Instances

There are quite a few interesting `Monoid` instances defined in `Data.Monoid`.

```
mempty `mappend` x = x
x `mappend` mempty = x
(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)
```

Listing 25: The `Monoid` laws

- `[a]` is a `Monoid`, with `mempty = []` and `mappend = (++)`. It is not hard to check that `(x ++ y) ++ z = x ++ (y ++ z)` for any lists `x`, `y`, and `z`, and that the empty list is the identity: `[] ++ x = x ++ [] = x`.
- As noted previously, we can make a monoid out of any numeric type under either addition or multiplication. However, since we can't have two instances for the same type, `Data.Monoid` provides two `newtype` wrappers, `Sum` and `Product`, with appropriate `Monoid` instances.

  ```
  > getSum (mconcat . map Sum $ [1..5])
  15
  > getProduct (mconcat . map Product $ [1..5])
  120
  ```

  This example code is silly, of course; we could just write `sum [1..5]` and `product [1..5]`. Nevertheless, these instances are useful in more generalized settings, as we will see in the discussion of `Foldable` (page 44).
- `Any` and `All` are `newtype` wrappers providing `Monoid` instances for `Bool` (under disjunction and conjunction, respectively).
- There are three instances for `Maybe`: a basic instance which lifts a `Monoid` instance for `a` to an instance for `Maybe a`, and two `newtype` wrappers `First` and `Last` for which `mappend` selects the first (respectively last) non-`Nothing` item.
- `Endo a` is a newtype wrapper for functions `a -> a`, which form a monoid under composition.
- There are several ways to "lift" `Monoid` instances to instances with additional structure. We have already seen that an instance for `a` can be lifted to an instance for `Maybe a`. There are also tuple instances: if `a` and `b` are instances of `Monoid`, then so is `(a,b)`, using the monoid operations for `a` and `b` in the obvious pairwise manner. Finally, if `a` is a `Monoid`, then so is the function type `e -> a` for any `e`; in particular, `g `mappend` h` is the function which applies both `g` and `h` to its argument and then combines the result using the underlying `Monoid` instance for `a`. This can be quite useful and elegant [72].
- The type `Ordering = LT | EQ | GT` is a `Monoid`, defined in such a way that `mconcat (zipWith compare xs ys)` computes the lexicographic ordering of `xs` and `ys`. In particular, `mempty = EQ`, and `mappend` evaluates to its leftmost

non-`EQ` argument (or `EQ` if both arguments are `EQ`). This can be used together with the function instance of `Monoid` to do some clever things [73].

- There are also `Monoid` instances for several standard data structures in the containers library [8], including `Map`, `Set`, and `Sequence`.

`Monoid` is also used to enable several other type class instances. As noted previously, we can use `Monoid` to make `((,) e)` an instance of `Applicative`, as shown in Listing 26.

```
instance Monoid e => Applicative ((,) e) where
  pure x = (mempty, x)
  (u, f) <*> (v, x) = (u `mappend` v, f x)
```

Listing 26: An `Applicative` instance for `((,) e)` using `Monoid`

`Monoid` can be similarly used to make `((,) e)` an instance of `Monad` as well; this is known as the **writer monad**. As we've already seen, `Writer` and `WriterT` are a newtype wrapper and transformer for this monad, respectively.

`Monoid` also plays a key role in the `Foldable` type class (page 44).

Other monoidal classes: Alternative, MonadPlus, ArrowPlus

The `Alternative` type class [74], shown in Listing 27, is for `Applicative` functors which also have a monoid structure.

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

Listing 27: The `Alternative` type class

Of course, instances of `Alternative` should satisfy the monoid laws.

Likewise, `MonadPlus` [75], shown in Listing 28, is for `Monad`s with a monoid structure.

The `MonadPlus` documentation states that it is intended to model monads which also support "choice and failure"; in addition to the monoid laws, instances of `MonadPlus` are expected to satisfy

```
mzero >>= f  =  mzero
v >> mzero   =  mzero
```

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

Listing 28: The `MonadPlus` type class

which explains the sense in which `mzero` denotes failure. Since `mzero` should be the identity for `mplus`, the computation `m1 ‘mplus‘ m2` succeeds (evaluates to something other than `mzero`) if either `m1` or `m2` does; so `mplus` represents choice. The `guard` function can also be used with instances of `MonadPlus`; it requires a condition to be satisfied and fails (using `mzero`) if it is not. A simple example of a `MonadPlus` instance is `[]`, which is exactly the same as the `Monoid` instance for `[]`: the empty list represents failure, and list concatenation represents choice. In general, however, a `MonadPlus` instance for a type need not be the same as its `Monoid` instance; `Maybe` is an example of such a type. A great introduction to the `MonadPlus` type class, with interesting examples of its use, is Doug Auclair's Monad.Reader article [76].

There used to be a type class called `MonadZero` containing only `mzero`, representing monads with failure. The `do`-notation requires some notion of failure to deal with failing pattern matches. Unfortunately, `MonadZero` was scrapped in favor of adding the `fail` method to the `Monad` class. If we are lucky, someday `MonadZero` will be restored, and `fail` will be banished to the bit bucket where it belongs [77]. The idea is that any `do`-block which uses pattern matching (and hence may fail) would require a `MonadZero` constraint; otherwise, only a `Monad` constraint would be required.

Finally, `ArrowZero` and `ArrowPlus` [78], shown in Listing 29, represent `Arrow`s (page 51) with a monoid structure.

```
class Arrow (~>) => ArrowZero (~>) where
  zeroArrow :: b ~> c

class ArrowZero (~>) => ArrowPlus (~>) where
  (<+>) :: (b ~> c) -> (b ~> c) -> (b ~> c)
```

Listing 29: The `ArrowZero` and `ArrowPlus` type classes

Further reading

Monoids have gotten a fair bit of attention recently, ultimately due to a blog post by Brian Hurt [79], in which he complained about the fact that the names of many Haskell type classes (`Monoid` in particular) are taken from abstract mathematics. This resulted in a long haskell-cafe thread [71] arguing the point and discussing monoids in general.

However, this was quickly followed by several blog posts about `Monoid`.[13] First, Dan Piponi wrote a great introductory post, "Haskell Monoids and their Uses" [80]. This was quickly followed by Heinrich Apfelmus's "Monoids and Finger Trees" [81], an accessible exposition of Hinze and Paterson's classic paper on 2-3 finger trees [82], which makes very clever use of `Monoid` to implement an elegant and generic data structure. Dan Piponi then wrote two fascinating articles about using `Monoids` (and finger trees) to perform fast incremental regular expression matching [83, 84].

In a similar vein, David Place's article on improving `Data.Map` in order to compute incremental folds [85] is also a good example of using `Monoid` to generalize a data structure.

Some other interesting examples of `Monoid` use include building elegant list sorting combinators [73], collecting unstructured information [86], and a brilliant series of posts by Chung-Chieh Shan and Dylan Thurston using `Monoid`s to elegantly solve a difficult combinatorial puzzle [87, 88, 89, 90].

As unlikely as it sounds, monads can actually be viewed as a sort of monoid, with `join` playing the role of the binary operation and `return` the role of the identity; see Dan Piponi's blog post [65].

Foldable

The `Foldable` class, defined in the `Data.Foldable` module [91], abstracts over containers which can be "folded" into a summary value. This allows such folding operations to be written in a container-agnostic way.

Definition

The definition of the `Foldable` type class is shown in Listing 30.

This may look complicated, but in fact, to make a `Foldable` instance you only need to implement one method: your choice of `foldMap` or `foldr`. All the other methods have default implementations in terms of these, and are presumably included in the class in case more efficient implementations can be provided.

[13] May its name live forever.

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

Listing 30: The `Foldable` type class

Instances and examples

The type of `foldMap` should make it clear what it is supposed to do: given a way to convert the data in a container into a `Monoid` (a function `a -> m`) and a container of `a`'s (`t a`), `foldMap` provides a way to iterate over the entire contents of the container, converting all the `a`'s to `m`'s and combining all the `m`'s with `mappend`. Listing 31 shows two examples: a simple implementation of `foldMap` for lists, and a binary tree example provided by the `Foldable` documentation.

```
instance Foldable [] where
  foldMap g = mconcat . map g

data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)

instance Foldable Tree where
  foldMap f Empty = mempty
  foldMap f (Leaf x) = f x
  foldMap f (Node l k r) = foldMap f l ++ f k ++ foldMap f r
    where (++) = mappend
```

Listing 31: Two `foldMap` examples

The `foldr` function has a type similar to the `foldr` found in the `Prelude`, but more general, since the `foldr` in the `Prelude` works only on lists.

The `Foldable` module also provides instances for `Maybe` and `Array`; additionally, many of the data structures found in the standard containers library [8] (for example, `Map`, `Set`, `Tree`, and `Sequence`) provide their own `Foldable` instances.

Derived folds

Given an instance of `Foldable`, we can write generic, container-agnostic functions such as the examples shown in Listing 32.

```
-- Compute the size of any container.
containerSize :: Foldable f => f a -> Int
containerSize = getSum . foldMap (const (Sum 1))

-- Compute a list of elements of a container satisfying a predicate.
filterF :: Foldable f => (a -> Bool) -> f a -> [a]
filterF p = foldMap (\a -> if p a then [a] else [])

-- Get a list of all the Strings in a container which include the
-- letter a.
aStrings :: Foldable f => f String -> [String]
aStrings = filterF (elem 'a')
```

Listing 32: `Foldable` examples

The `Foldable` module also provides a large number of predefined folds, many of which are generalized versions of `Prelude` functions of the same name that only work on lists: `concat`, `concatMap`, `and`, `or`, `any`, `all`, `sum`, `product`, `maximum`(`By`), `minimum`(`By`), `elem`, `notElem`, and `find`. The reader may enjoy coming up with elegant implementations of these functions using `fold` or `foldMap` and appropriate `Monoid` instances.

There are also generic functions that work with `Applicative` or `Monad` instances to generate some sort of computation from each element in a container, and then perform all the side effects from those computations, discarding the results: `traverse_`, `sequenceA_`, and others. The results must be discarded because the `Foldable` class is too weak to specify what to do with them: we cannot, in general, make an arbitrary `Applicative` or `Monad` instance into a `Monoid`. If we do have an `Applicative` or `Monad` with a monoid structure—that is, an `Alternative` or a `MonadPlus`—then we can use the `asum` or `msum` functions, which can combine the results as well. Consult the `Foldable` documentation [91] for more details on any of these functions.

Note that the `Foldable` operations always forget the structure of the container being folded. If we start with a container of type `t a` for some `Foldable t`, then `t` will never appear in the output type of any operations defined in the `Foldable` module. Many times this is exactly what we want, but sometimes we would like

to be able to generically traverse a container while preserving its structure—and this is exactly what the `Traversable` class provides, which will be discussed in the next section.

Further reading

The `Foldable` class had its genesis in McBride and Paterson's paper introducing `Applicative` [12], although it has been fleshed out quite a bit from the form in the paper.

An interesting use of `Foldable` (as well as `Traversable`) can be found in Janis Voigtländer's paper **Bidirectionalization for free!** [92].

Traversable

Definition

The `Traversable` type class, defined in the `Data.Traversable` module [93], is shown in Listing 33.

```
class (Functor t, Foldable t) => Traversable t where
  traverse  :: Applicative f => (a -> f b) -> t a -> f (t b)
  sequenceA :: Applicative f => t (f a) -> f (t a)
  mapM      :: Monad m => (a -> m b) -> t a -> m (t b)
  sequence  :: Monad m => t (m a) -> m (t a)
```

Listing 33: The `Traversable` type class

As you can see, every `Traversable` is also a foldable functor. Like `Foldable`, there is a lot in this type class, but making instances is actually rather easy: one need only implement `traverse` or `sequenceA`; the other methods all have default implementations in terms of these functions. A good exercise is to figure out what the default implementations should be: given either `traverse` or `sequenceA`, how would you define the other three methods? (Hint for `mapM`: `Control.Applicative` exports the `WrapMonad` newtype, which makes any `Monad` into an `Applicative`. The `sequence` function can be implemented in terms of `mapM`.)

Intuition

The key method of the `Traversable` class, and the source of its unique power, is `sequenceA`. Consider its type:

```
sequenceA :: Applicative f => t (f a) -> f (t a)
```

This answers the fundamental question: when can we commute two functors? For example, can we turn a tree of lists into a list of trees? (Answer: yes, in two ways. Figuring out what they are, and why, is left as an exercise. A much more challenging question is whether a list of trees can be turned into a tree of lists.)

The ability to compose two monads depends crucially on this ability to commute functors. Intuitively, if we want to build a composed monad `M a = m (n a)` out of monads `m` and `n`, then to be able to implement `join :: M (M a) -> M a`, that is, `join :: m (n (m (n a))) -> m (n a)`, we have to be able to commute the `n` past the `m` to get `m (m (n (n a)))`, and then we can use the `join`s for `m` and `n` to produce something of type `m (n a)`. See Mark Jones's paper for more details [42].

Instances and examples

What's an example of a `Traversable` instance? Listing 34 shows an example instance for the same `Tree` type used as an example in the previous `Foldable` section. It is instructive to compare this instance with a `Functor` instance for `Tree`, which is also shown.

```
data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)

instance Traversable Tree where
  traverse g Empty        = pure Empty
  traverse g (Leaf x)     = Leaf <$> g x
  traverse g (Node l x r) = Node <$> traverse g l
                                 <*> g x
                                 <*> traverse g r

instance Functor Tree where
  fmap     g Empty        = Empty
  fmap     g (Leaf x)     = Leaf $ g x
  fmap     g (Node l x r) = Node (fmap g l)
                                 (g x)
                                 (fmap g r)
```

Listing 34: An example `Tree` instance of `Traversable`

It should be clear that the `Traversable` and `Functor` instances for `Tree` are almost identical; the only difference is that the `Functor` instance involves normal

function application, whereas the applications in the `Traversable` instance take place within an `Applicative` context, using `(<$>)` and `(<*>)`. In fact, this will be true for any type.

Any `Traversable` functor is also `Foldable`, and a `Functor`. We can see this not only from the class declaration, but by the fact that we can implement the methods of both classes given only the `Traversable` methods. A good exercise is to implement `fmap` and `foldMap` using only the `Traversable` methods; the implementations are surprisingly elegant. The `Traversable` module provides these implementations as `fmapDefault` and `foldMapDefault`.

The standard libraries provide a number of `Traversable` instances, including instances for `[]`, `Maybe`, `Map`, `Tree`, and `Sequence`. Notably, `Set` is not `Traversable`, although it is `Foldable`.

Further reading

The `Traversable` class also had its genesis in McBride and Paterson's `Applicative` paper [12], and is described in more detail in Gibbons and Oliveira, **The Essence of the Iterator Pattern** [94], which also contains a wealth of references to related work.

Category

`Category` is another fairly new addition to the Haskell standard libraries; you may or may not have it installed depending on the version of your `base` package. It generalizes the notion of function composition to general "morphisms."

The definition of the `Category` type class (from `Control.Category` [95]) is shown in Listing 35. For ease of reading, note that I have used an infix type constructor `(~>)`, much like the infix function type constructor `(->)`. This syntax is not part of Haskell 98. The second definition shown is the one used in the standard libraries. For the remainder of the article, I will use the infix type constructor `(~>)` for `Category` as well as `Arrow`.

Note that an instance of `Category` should be a type constructor which takes two type arguments, that is, something of kind `* -> * -> *`. It is instructive to imagine the type constructor variable `cat` replaced by the function constructor `(->)`: indeed, in this case we recover precisely the familiar identity function `id` and function composition operator `(.)` defined in the standard `Prelude`.

Of course, the `Category` module provides exactly such an instance of `Category` for `(->)`. But it also provides one other instance, shown in Listing 36, which should be familiar from the previous discussion of the `Monad` laws. `Kleisli m a b`, as defined in the `Control.Arrow` module, is just a `newtype` wrapper around `a -> m b`.

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

-- The same thing, with a normal (prefix) type constructor
class Category cat where
  id  :: cat a a
  (.) :: cat b c -> cat a b -> cat a c
```

Listing 35: The `Category` type class

```
newtype Kleisli m a b = Kleisli { runKleisli :: a -> m b }

instance Monad m => Category (Kleisli m) where
  id = Kleisli return
  Kleisli g . Kleisli h = Kleisli (h >=> g)
```

Listing 36: The `Kleisli Category` instance

The only law that `Category` instances should satisfy is that `id` and `(.)` should form a monoid—that is, `id` should be the identity of `(.)`, and `(.)` should be associative.

Finally, the `Category` module exports two additional operators: `(<<<)`, which is just a synonym for `(.)`, and `(>>>)`, which is `(.)` with its arguments reversed. (In previous versions of the libraries, these operators were defined as part of the `Arrow` class.)

Further reading

The name `Category` is a bit misleading, since the `Category` class cannot represent arbitrary categories, but only categories whose objects are objects of `Hask`, the category of Haskell types. For a more general treatment of categories within Haskell, see the category-extras package [2]. For more about category theory in general, see the excellent Haskell wikibook page [9], Steve Awodey's new book [96], Benjamin Pierce's **Basic category theory for computer scientists** [97], or Barr and Wells's category theory lecture notes [98]. Benjamin Russell's blog post [99] is another good source of motivation and category theory links. You certainly don't need to know any category theory to be a successful and productive Haskell programmer,

but it does lend itself to much deeper appreciation of Haskell's underlying theory.

Arrow

The `Arrow` class represents another abstraction of computation, in a similar vein to `Monad` and `Applicative`. However, unlike `Monad` and `Applicative`, whose types only reflect their output, the type of an `Arrow` computation reflects both its input and output. Arrows generalize functions: if `(~>)` is an instance of `Arrow`, a value of type `b ~> c` can be thought of as a computation which takes values of type `b` as input, and produces values of type `c` as output. In the `(->)` instance of `Arrow` this is just a pure function; in general, however, an arrow may represent some sort of "effectful" computation.

Definition

The definition of the `Arrow` type class, from `Control.Arrow` [100], is shown in Listing 37.

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

Listing 37: The `Arrow` type class

The first thing to note is the `Category` class constraint, which means that we get identity arrows and arrow composition for free: given two arrows `g :: b ~> c` and `h :: c ~> d`, we can form their composition `g >>> h :: b ~> d`.[14]

As should be a familiar pattern by now, the only methods which must be defined when writing a new instance of `Arrow` are `arr` and `first`; the other methods have default definitions in terms of these, but are included in the `Arrow` class so that they can be overridden with more efficient implementations if desired.

[14] In versions of the `base` package prior to version 4, there is no `Category` class, and the `Arrow` class includes the arrow composition operator `(>>>)`. It also includes `pure` as a synonym for `arr`, but this was removed since it conflicts with the `pure` from `Applicative`.

Intuition

Let's look at each of the arrow methods in turn. Ross Paterson's web page on arrows [101] has nice diagrams which can help build intuition.

- The `arr` function takes any function `b -> c` and turns it into a generalized arrow `b ~> c`. The `arr` method justifies the claim that arrows generalize functions, since it says that we can treat any function as an arrow. It is intended that the arrow `arr g` is "pure" in the sense that it only computes `g` and has no "effects" (whatever that might mean for any particular arrow type).
- The `first` method turns any arrow from `b` to `c` into an arrow from `(b,d)` to `(c,d)`. The idea is that `first g` uses `g` to process the first element of a tuple, and lets the second element pass through unchanged. For the function instance of `Arrow`, of course, `first g (x,y) = (g x, y)`.
- The `second` function is similar to `first`, but with the elements of the tuples swapped. Indeed, it can be defined in terms of `first` using an auxiliary function `swap`, defined by `swap (x,y) = (y,x)`.
- The `(***)` operator is "parallel composition" of arrows: it takes two arrows and makes them into one arrow on tuples, which has the behavior of the first arrow on the first element of a tuple, and the behavior of the second arrow on the second element. The mnemonic is that `g *** h` is the **product** (hence `*`) of `g` and `h`. For the function instance of `Arrow`, we define `(g *** h) (x,y) = (g x, h y)`. The default implementation of `(***)` is in terms of `first`, `second`, and sequential arrow composition `(>>>)`. The reader may also wish to think about how to implement `first` and `second` in terms of `(***)`.
- The `(&&&)` operator is "fanout composition" of arrows: it takes two arrows `g` and `h` and makes them into a new arrow `g &&& h` which supplies its input as the input to both `g` and `h`, returning their results as a tuple. The mnemonic is that `g &&& h` performs both `g` **and** `h` (hence `&`) on its input. For functions, we define `(g &&& h) x = (g x, h x)`.

Instances

The `Arrow` library itself only provides two `Arrow` instances, both of which we have already seen: `(->)`, the normal function constructor, and `Kleisli m`, which makes functions of type `a -> m b` into `Arrow`s for any `Monad m`. These instances are shown in Listing 38.

```
instance Arrow (->) where
  arr g = g
  first g (x,y) = (g x, y)

newtype Kleisli m a b = Kleisli { runKleisli :: a -> m b }

instance Monad m => Arrow (Kleisli m) where
  arr f = Kleisli (return . f)
  first (Kleisli f) = Kleisli (\ ~(b,d) -> do c <- f b
                                              return (c,d) )
```

Listing 38: The `(->)` and `Kleisli m` instances of `Arrow`

Laws

There are quite a few laws that instances of `Arrow` should satisfy [102, 103, 104]; they are shown in Listing 39. Note that the version of the laws shown in Listing 39 is slightly different than the laws given in the first two above references, since several of the laws have now been subsumed by the `Category` laws (in particular, the requirements that `id` is the identity arrow and that `(>>>)` is associative). The laws shown here follow those in Paterson [104], which uses the `Category` class.

```
                      arr id  =  id
                arr (h . g)   =  arr g >>> arr h
              first (arr g)   =  arr (g *** id)
            first (g >>> h)   =  first g >>> first h
  first g >>> arr (id *** h)  =  arr (id *** h) >>> first g
         first g >>> arr fst  =  arr fst >>> g
first (first g) >>> arr assoc =  arr assoc >>> first g

assoc ((x,y),z) = (x,(y,z))
```

Listing 39: The `Arrow` laws

The reader is advised not to lose too much sleep over the `Arrow` laws,[15] since it is not essential to understand them in order to program with arrows. There are also laws that `ArrowChoice`, `ArrowApply`, and `ArrowLoop` instances should satisfy; the interested reader should consult Paterson [104].

[15] Unless category-theory-induced insomnolence is your cup of tea.

ArrowChoice

Computations built using the `Arrow` class, like those built using the `Applicative` class, are rather inflexible: the structure of the computation is fixed at the outset, and there is no ability to choose between alternate execution paths based on intermediate results. The `ArrowChoice` class provides exactly such an ability; it is shown in Listing 40.

```
class Arrow (~>) => ArrowChoice (~>) where
  left  :: (b ~> c) -> (Either b d ~> Either c d)
  right :: (b ~> c) -> (Either d b ~> Either d c)
  (+++) :: (b ~> c) -> (b' ~> c') -> (Either b b' ~> Either c c')
  (|||) :: (b ~> d) -> (c ~> d) -> (Either b c ~> d)
```

Listing 40: The `ArrowChoice` type class

A comparison of `ArrowChoice` to `Arrow` will reveal a striking parallel between `left`, `right`, `(+++)`, `(|||)` and `first`, `second`, `(***)`, `(&&&)`, respectively. Indeed, they are dual: `first`, `second`, `(***)`, and `(&&&)` all operate on product types (tuples), and `left`, `right`, `(+++)`, and `(|||)` are the corresponding operations on sum types. In general, these operations create arrows whose inputs are tagged with `Left` or `Right`, and can choose how to act based on these tags.

- If `g` is an arrow from `b` to `c`, then `left g` is an arrow from `Either b d` to `Either c d`. On inputs tagged with `Left`, the `left g` arrow has the behavior of `g`; on inputs tagged with `Right`, it behaves as the identity.
- The `right` function, of course, is the mirror image of `left`. The arrow `right g` has the behavior of `g` on inputs tagged with `Right`.
- The `(+++)` operator performs "multiplexing": `g +++ h` behaves as `g` on inputs tagged with `Left`, and as `h` on inputs tagged with `Right`. The tags are preserved. The `(+++)` operator is the **sum** (hence `+`) of two arrows, just as `(***)` is the product.
- The `(|||)` operator is "merge" or "fanin": the arrow `g ||| h` behaves as `g` on inputs tagged with `Left`, and `h` on inputs tagged with `Right`, but the tags are discarded (hence, `g` and `h` must have the same output type). The mnemonic is that `g ||| h` performs either `g` **or** `h` on its input.

The `ArrowChoice` class allows computations to choose among a finite number of execution paths, based on intermediate results. The possible execution paths must be known in advance, and explicitly assembled with `(+++)` or `(|||)`. However, sometimes more flexibility is needed: we would like to be able to **compute** an arrow from intermediate results, and use this computed arrow to continue the computation. This is the power given to us by `ArrowApply`.

ArrowApply

The `ArrowApply` type class is shown in Listing 41.

```
class Arrow (~>) => ArrowApply (~>) where
  app :: (b ~> c, b) ~> c
```

Listing 41: The `ArrowApply` type class

If we have computed an arrow as the output of some previous computation, then `app` allows us to apply that arrow to an input, producing its output as the output of `app`. As an exercise, the reader may wish to use `app` to implement an alternative "curried" version, `app2 :: b ~> ((b ~> c) ~> c)`.

This notion of being able to **compute** a new computation may sound familiar: this is exactly what the monadic bind operator `(>>=)` does. It should not particularly come as a surprise that `ArrowApply` and `Monad` are exactly equivalent in expressive power. In particular, `Kleisli m` can be made an instance of `ArrowApply`, and any instance of `ArrowApply` can be made a `Monad` (via the `newtype` wrapper `ArrowMonad`). As an exercise, the reader may wish to try implementing these instances, shown in Listing 42.

```
instance Monad m => ArrowApply (Kleisli m) where
  app =    -- exercise

newtype ArrowApply a => ArrowMonad a b = ArrowMonad (a () b)

instance ArrowApply a => Monad (ArrowMonad a) where
  return               =    -- exercise
  (ArrowMonad a) >>= k =    -- exercise
```

Listing 42: Equivalence of `ArrowApply` and `Monad`

ArrowLoop

The `ArrowLoop` type class is shown in Listing 43; it describes arrows that can use recursion to compute results, and is used to desugar the `rec` construct in arrow notation (described below).

Taken by itself, the type of the `loop` method does not seem to tell us much. Its intention, however, is a generalization of the `trace` function which is also shown.

The `d` component of the first arrow's output is fed back in as its own input. In other words, the arrow `loop g` is obtained by recursively "fixing" the second component of the input to `g`.

```
class Arrow a => ArrowLoop a where
  loop :: a (b, d) (c, d) -> a b c

trace :: ((b,d) -> (c,d)) -> b -> c
trace f b = let (c,d) = f (b,d) in c
```

Listing 43: The `ArrowLoop` type class

It can be a bit difficult to grok what the `trace` function is doing. How can `d` appear on the left and right sides of the `let`? Well, this is Haskell's laziness at work. There is not space here for a full explanation; the interested reader is encouraged to study the standard `fix` function, and to read Paterson's arrow tutorial [104].

Arrow notation

Programming directly with the arrow combinators can be painful, especially when writing complex computations which need to retain simultaneous reference to a number of intermediate results. With nothing but the arrow combinators, such intermediate results must be kept in nested tuples, and it is up to the programmer to remember which intermediate results are in which components, and to swap, reassociate, and generally mangle tuples as necessary. This problem is solved by the special arrow notation supported by GHC, similar to `do` notation for monads, that allows names to be assigned to intermediate results while building up arrow computations. An example arrow implemented using arrow notation, taken from Paterson [104], is shown in Listing 44. This arrow is intended to represent a recursively defined counter circuit with a reset line.

There is not space here for a full explanation of arrow notation; the interested reader should consult Paterson's paper introducing the notation [105], or his later tutorial which presents a simplified version [104].

Further reading

An excellent starting place for the student of arrows is the web page put together by Paterson [101], which contains an introduction and many references. Some key papers on arrows include Hughes's original paper introducing arrows, **Generalising Monads to Arrows** [102], and Paterson's paper on arrow notation [105].

```
class ArrowLoop (~>) => ArrowCircuit (~>) where
  delay :: b -> (b ~> b)

counter :: ArrowCircuit (~>) => Bool ~> Int
counter = proc reset -> do
            rec output <- idA     -< if reset then 0 else next
                next   <- delay 0 -< output + 1
            idA -< output
```

Listing 44: An example arrow using arrow notation

Both Hughes and Paterson later wrote accessible tutorials intended for a broader audience [104, 106].

Although Hughes's goal in defining the `Arrow` class was to generalize `Monad`s, and it has been said that `Arrow` lies "between `Applicative` and `Monad`" in power, they are not directly comparable. The precise relationship remained in some confusion until analyzed by Lindley, Wadler, and Yallop [107], who also invented a new calculus of arrows, based on the lambda calculus, which considerably simplifies the presentation of the arrow laws [103].

Some examples of `Arrow`s include Yampa [108], the Haskell XML Toolkit [109], and the functional GUI library Grapefruit [110].

Some extensions to arrows have been explored; for example, the `BiArrow`s of Alimarine et al., for two-way instead of one-way computation [111].

Links to many additional research papers relating `Arrow`s can be found on the Haskell wiki [69].

Comonad

The final type class we will examine is `Comonad`. The `Comonad` class is the categorical dual of `Monad`; that is, `Comonad` is like `Monad` but with all the function arrows flipped. It is not actually in the standard Haskell libraries, but it has seen some interesting uses recently, so we include it here for completeness.

Definition

The `Comonad` type class, defined in the `Control.Comonad` module of the category-extras library [2], is shown in Listing 45.

As you can see, `extract` is the dual of `return`, `duplicate` is the dual of `join`, and `extend` is the dual of `(>>=)` (although its arguments are in a different order).

```
class Functor f => Copointed f where
  extract :: f a -> a

class Copointed w => Comonad w where
  duplicate :: w a -> w (w a)
  extend :: (w a -> b) -> w a -> w b
```

Listing 45: The `Comonad` type class

The definition of `Comonad` is a bit redundant (after all, the `Monad` class does not need `join`), but this is so that a `Comonad` can be defined by `fmap`, `extract`, and **either** `duplicate` or `extend`. Each has a default implementation in terms of the other.

A prototypical example of a `Comonad` instance is shown in Listing 46.

```
-- Infinite lazy streams
data Stream a = Cons a (Stream a)

instance Functor Stream where
  fmap g (Cons x xs) = Cons (g x) (fmap g xs)

instance Copointed Stream where
  extract (Cons x _) = x

-- 'duplicate' is like the list function 'tails'
-- 'extend' computes a new Stream from an old, where the element
--   at position n is computed as a function of everything from
--   position n onwards in the old Stream
instance Comonad Stream where
  duplicate s@(Cons x xs) = Cons s (duplicate xs)
  extend g s@(Cons x xs)  = Cons (g s) (extend g xs)
                       -- = fmap g (duplicate s)
```

Listing 46: A `Comonad` instance for `Stream`

Further reading

Dan Piponi explains in a blog post what cellular automata have to do with comonads [112]. In another blog post, Conal Elliott has examined a comonadic formulation of functional reactive programming [113]. Sterling Clover's blog post **Comonads in everyday life** [114] explains the relationship between comonads and zippers, and how comonads can be used to design a menu system for a web site.

Uustalu and Vene have a number of papers exploring ideas related to comonads and functional programming [115, 116, 117, 118, 119].

Acknowledgements

A special thanks to all of those who taught me about standard Haskell type classes and helped me develop good intuition for them, particularly Jules Bean (quicksilver), Derek Elkins (ddarius), Conal Elliott (conal), Cale Gibbard (Cale), David House, Dan Piponi (sigfpe), and Kevin Reid (kpreid).

I also thank the many people who provided a mountain of helpful feedback and suggestions on a first draft of this article: David Amos, Kevin Ballard, Reid Barton, Doug Beardsley, Joachim Breitner, Andrew Cave, David Christiansen, Gregory Collins, Mark Jason Dominus, Conal Elliott, Yitz Gale, George Giorgidze, Steven Grady, Travis Hartwell, Steve Hicks, Philip Hölzenspies, Edward Kmett, Eric Kow, Serge Le Huitouze, Felipe Lessa, Stefan Ljungstrand, Eric Macaulay, Rob MacAulay, Simon Meier, Eric Mertens, Tim Newsham, Russell O'Connor, Conrad Parker, Walt Rorie-Baety, Colin Ross, Tom Schrijvers, Aditya Siram, C. Smith, Martijn van Steenbergen, Joe Thornber, Jared Updike, Rob Vollmert, Andrew Wagner, Louis Wasserman, and Ashley Yakeley, as well as a few only known to me by their IRC nicks: b_jonas, maltem, tehgeekmeister, and ziman. I have undoubtedly omitted a few inadvertently, which in no way diminishes my gratitude.

Finally, I would like to thank Wouter Swierstra for his fantastic work editing the Monad.Reader, and my wife Joyia for her patience during the process of writing the Typeclassopedia.

About the author

Brent Yorgey [120, 121] is a first-year Ph.D. student in the programming languages group at the University of Pennsylvania [122]. He enjoys teaching, creating EDSLs, playing Bach fugues, musing upon category theory, and cooking tasty lambda-treats for the denizens of #haskell.

References

[1] Brent Yorgey. Abstraction, intuition, and the "monad tutorial fallacy". http://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/.

[2] Edward A. Kmett, Dave Menendez, and Iavor Diatchki. The category-extras library. http://hackage.haskell.org/cgi-bin/hackage-scripts/package/category-extras.

[3] Simon Peyton Jones (ed.). Haskell 98 language and libraries revised report. **J. Funct. Program.**, 13(1) (2003). http://haskell.org/onlinereport/.

[4] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. **ACM Trans. Program. Lang. Syst.**, 18(2):pages 109–138 (1996).

[5] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In **Proceedings of the 1997 Haskell Workshop**. ACM (1997).

[6] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In **HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages**, pages 12–1–12–55. ACM, New York, NY, USA (2007).

[7] Functor documentation. http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#t%3AFunctor.

[8] The containers library. http://hackage.haskell.org/packages/archive/containers/0.2.0.0/doc/html/index.html.

[9] Haskell wikibook: Category theory. http://en.wikibooks.org/wiki/Haskell/Category_theory.

[10] Philip Wadler. Theorems for free! In **FPCA '89: Proceedings of the fourth international conference on functional programming languages and computer architecture**, pages 347–359. ACM, New York, NY, USA (1989).

[11] Applicative documentation. http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Applicative.html.

[12] Conor McBride and Ross Paterson. Applicative programming with effects. **J. Funct. Program.**, 18(1):pages 1–13 (2008). http://www.soi.city.ac.uk/~ross/papers/Applicative.html.

[13] Philip Wadler. How to replace failure by a list of successes. In **Proc. of a conference on Functional programming languages and computer architecture**, pages 113–128. Springer-Verlag New York, Inc., New York, NY, USA (1985).

[14] Conal Elliott. Functional Images. In Jeremy Gibbons and Oege de Moor (editors), **The Fun of Programming**. "Cornerstones of Computing" series, Palgrave (March 2003). http://conal.net/papers/functional-images/.

[15] Conal Elliott. Simply efficient functional reactivity. Technical Report 2008-01, LambdaPix (2008). http://conal.net/papers/simply-reactive/.

[16] Conal Elliott. Posts with tag "applicative-functor". http://conal.net/blog/tag/applicative-functor.

[17] Conal Elliott. The TypeCompose library. http://haskell.org/haskellwiki/TypeCompose.

[18] Daan Leijen. Parsec. http://legacy.cs.uu.nl/daan/parsec.html.

[19] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Departement of Computer Science, Universiteit Utrecht (2001). http://legacy.cs.uu.nl/daan/download/papers/parsec-paper.pdf.

[20] Bryan O'Sullivan. The basics of applicative functors, put to practical work. http://www.serpentine.com/blog/2008/02/06/the-basics-of-applicative-functors-put-to-practical-work/.

[21] Chris Done. Applicative and ConfigFile, HSQL. http://chrisdone.com/blog/html/2009-02-10-applicative-configfile-hsql.html.

[22] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. An idiom's guide to formlets. Technical report, University of Edinburgh (2008). http://groups.inf.ed.ac.uk/links/formlets/.

[23] Monad documentation. http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html#9.

[24] The monad transformer library (mtl). http://hackage.haskell.org/cgi-bin/hackage-scripts/package/mtl.

[25] Identity documentation. http://hackage.haskell.org/packages/archive/mtl/1.1.0.2/doc/html/Control-Monad-Identity.html.

[26] Dan Piponi. The Trivial Monad. http://blog.sigfpe.com/2007/04/trivial-monad.html.

[27] Control.Monad.Reader documentation. http://hackage.haskell.org/packages/archive/mtl/1.1.0.2/doc/html/Control-Monad-Reader.html.

[28] Control.Monad.Writer documentation. http://hackage.haskell.org/packages/archive/mtl/1.1.0.2/doc/html/Control-Monad-Writer-Lazy.html.

[29] Control.Monad.State documentation. http://hackage.haskell.org/packages/archive/mtl/1.1.0.2/doc/html/Control-Monad-State-Lazy.html.

[30] Control.Monad.Cont documentation. http://hackage.haskell.org/packages/archive/mtl/1.1.0.2/doc/html/Control-Monad-Cont.html.

[31] Dan Piponi. The Mother of all Monads. http://blog.sigfpe.com/2008/12/mother-of-all-monads.html.

[32] Control.Monad documentation. http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Monad.html.

[33] Henk-Jan van Tuyl. A tour of the Haskell Monad functions. http://members.chello.nl/hjgtuyl/tourdemonad.html.

[34] Jeff Heard and Wren Thornton. The IfElse package. http://hackage.haskell.org/cgi-bin/hackage-scripts/package/IfElse.

[35] Monad laws. http://haskell.org/haskellwiki/Monad_laws.

[36] Control.Monad.Error documentation. http://hackage.haskell.org/packages/archive/mtl/1.1.0.2/doc/html/Control-Monad-Error.html.

[37] Martin Grabmüller. Monad Transformers Step by Step (October 2006). http://user.cs.tu-berlin.de/~magr/pub/Transformers.en.html. Draft paper.

[38] Cale Gibbard. How To Use Monad Transformers. http://cale.yi.org/index.php/How_To_Use_Monad_Transformers.

[39] Dan Piponi. Grok Haskell Monad Transformers. http://blog.sigfpe.com/2006/05/grok-haskell-monad-transformers.html.

[40] Levent Erkok. **Value recursion in monadic computations**. Ph.D. thesis (2002). Adviser-Launchbury,, John.

[41] Philip Wadler. The essence of functional programming. In **POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages**, pages 1–14. ACM, New York, NY, USA (1992). http://homepages.inf.ed.ac.uk/wadler/topics/monads.html.

[42] Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In **Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text**, pages 97–136. Springer-Verlag, London, UK (1995). http://web.cecs.pdx.edu/~mpj/pubs/springschool.html.

[43] Jeff Newbern. All About Monads. http://www.haskell.org/all_about_monads/html/.

[44] Cale Gibbard. Monads as containers.
http://haskell.org/haskellwiki/Monads_as_Containers.

[45] Eric Kow. Understanding monads. http://en.wikibooks.org/w/index.php?title=Haskell/Understanding_monads&oldid=933545.

[46] Andrea Rossato. The Monadic Way.
http://haskell.org/haskellwiki/The_Monadic_Way.

[47] Dan Piponi. You Could Have Invented Monads! (And Maybe You Already Have.). http://blog.sigfpe.com/2006/08/you-could-have-invented-monads-and.html.

[48] Andrew Pimlott. there's a monster in my Haskell! http://www.haskell.org/pipermail/haskell-cafe/2006-November/019190.html.

[49] Karsten Wagner. Understanding Monads. For real. http://kawagner.blogspot.com/2007/02/understanding-monads-for-real.html.

[50] Eric Kidd. Monads in 15 minutes: Backtracking and Maybe.
http://www.randomhacks.net/articles/2007/03/12/monads-in-15-minutes.

[51] Cale Gibbard. Monads as computation.
http://haskell.org/haskellwiki/Monads_as_computation.

[52] Richard Smith. Practical Monads.
http://metafoo.co.uk/practical-monads.txt.

[53] Cale Gibbard. Introduction to IO.
http://www.haskell.org/haskellwiki/Introduction_to_IO.

[54] Monad tutorials timeline.
http://haskell.org/haskellwiki/Monad_tutorials_timeline.

[55] Don Stewart and Eric Kow. think of a monad...
http://koweycode.blogspot.com/2007/01/think-of-monad.html.

[56] C. S. Gordon. Monads! (and Why Monad Tutorials Are All Awful).
http://ahamsandwich.wordpress.com/2007/07/26/monads-and-why-monad-tutorials-are-all-awful/.

[57] Dan Piponi. Monads, a Field Guide.
http://blog.sigfpe.com/2006/10/monads-field-guide.html.

[58] Tim Newsham. What's a Monad?
http://www.thenewsh.com/~newsham/haskell/monad.html.

[59] Blog articles/Monads.
http://haskell.org/haskellwiki/Blog_articles/Monads.

[60] Eric Kidd. How to make Data.Set a monad. http://www.randomhacks.net/articles/2007/03/15/data-set-monad-haskell-macros.

[61] Ganesh Sittampalam and Peter Gavin. The rmonad library. http://hackage.haskell.org/cgi-bin/hackage-scripts/package/rmonad.

[62] Do notation considered harmful. http://haskell.org/haskellwiki/Do_notation_considered_harmful.

[63] Robert Atkey. Parameterised Notions of Computation. **J. Funct. Prog.** (2008). http://homepages.inf.ed.ac.uk/ratkey/paramnotions-jfp.pdf.

[64] Dan Piponi. Beyond Monads. http://blog.sigfpe.com/2009/02/beyond-monads.html.

[65] Dan Piponi. From Monoids to Monads. http://blog.sigfpe.com/2008/11/from-monoids-to-monads.html.

[66] Mark Dominus. Triples and Closure. http://blog.plover.com/math/monad-closure.html.

[67] Derek Elkins. Calculating monads with category theory. **The Monad.Reader**, (13) (2009).

[68] Christoph Lüth and Neil Ghani. Composing monads using coproducts. **SIGPLAN Not.**, 37(9):pages 133–144 (2002).

[69] Research papers: Monads and Arrows. http://haskell.org/haskellwiki/Research_papers/Monads_and_arrows.

[70] Monoid documentation. http://haskell.org/ghc/docs/latest/html/libraries/base/Data-Monoid.html.

[71] haskell-cafe thread: Comments from OCaml Hacker Brian Hurt. http://thread.gmane.org/gmane.comp.lang.haskell.cafe/50590.

[72] Wouter Swierstra. Reply to "Looking for pointfree version". http://thread.gmane.org/gmane.comp.lang.haskell.cafe/52416.

[73] Cale Gibbard. Comment on "Monoids? In my programming language?". http://www.reddit.com/r/programming/comments/7cf4r/monoids_in_my_programming_language/c06adnx.

[74] Alternative documentation. http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Applicative.html#2.

[75] MonadPlus documentation. http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Monad.html#t%3AMonadPlus.

[76] Douglas M. Auclair. MonadPlus: What a Super Monad! **The Monad.Reader**, (11):pages 39–48 (August 2008). http://www.haskell.org/sitewiki/images/6/6a/TMR-Issue11.pdf.

[77] MonadPlus reform proposal. http://www.haskell.org/haskellwiki/MonadPlus_reform_proposal.

[78] ArrowZero and ArrowPlus documentation. http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Arrow.html#t%3AArrowZero.

[79] Brian Hurt. Random thoughts on haskell. http://enfranchisedmind.com/blog/2009/01/15/random-thoughts-on-haskell/.

[80] Dan Piponi. Haskell Monoids and their Uses. http://blog.sigfpe.com/2009/01/haskell-monoids-and-their-uses.html.

[81] Heinrich Apfelmus. Monoids and Finger Trees. http://apfelmus.nfshost.com/monoid-fingertree.html.

[82] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. **J. Funct. Program.**, 16(2):pages 197–217 (2006). http://www.soi.city.ac.uk/%7Eross/papers/FingerTree.html.

[83] Dan Piponi. Fast incremental regular expression matching with monoids. http://blog.sigfpe.com/2009/01/fast-incremental-regular-expression.html.

[84] Dan Piponi. Beyond Regular Expressions: More Incremental String Matching. http://blog.sigfpe.com/2009/01/beyond-regular-expressions-more.html.

[85] David F. Place. How to Refold a Map. **The Monad.Reader**, (11):pages 5–14 (August 2008). http://www.haskell.org/sitewiki/images/6/6a/TMR-Issue11.pdf.

[86] Brent Yorgey. Collecting unstructured information with the monoid of partial knowledge. http://byorgey.wordpress.com/2008/04/17/collecting-unstructured-information-with-the-monoid-of-partial-knowledge/.

[87] Chung-Chieh Shan and Dylan Thurston. Word numbers, part 1: Billion approaches. http://conway.rutgers.edu/~ccshan/wiki/blog/posts/WordNumbers1/.

[88] Chung-Chieh Shan and Dylan Thurston. Word numbers, part 2. http://conway.rutgers.edu/~ccshan/wiki/blog/posts/WordNumbers2/.

[89] Chung-Chieh Shan and Dylan Thurston. Word numbers, part 3: Binary search. http://conway.rutgers.edu/~ccshan/wiki/blog/posts/WordNumbers3/.

[90] Chung-Chieh Shan and Dylan Thurston. Word numbers, part 4: Sort the words, sum the numbers. http://conway.rutgers.edu/~ccshan/wiki/blog/posts/WordNumbers4/.

[91] Foldable documentation. http://haskell.org/ghc/docs/latest/html/libraries/base/Data-Foldable.html.

[92] Janis Voigtländer. Bidirectionalization for free! (pearl). In **POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages**, pages 165–176. ACM, New York, NY, USA (2009).

[93] Traversable documentation. http://haskell.org/ghc/docs/latest/html/libraries/base/Data-Traversable.html.

[94] Jeremy Gibbons and Bruno C. d. S. Oliveira. The essence of the Iterator pattern. **Submitted for publication** (2007). http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/iterator.pdf.

[95] Category documentation. http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Category.html.

[96] Steve Awodey. **Category Theory**. Clarendon Press (2006).

[97] Benjamin C. Pierce. **Basic category theory for computer scientists**. MIT Press, Cambridge, MA, USA (1991).

[98] Michael Barr and Charles Wells. Category Theory. In **Lecture Notes of the 11th European Summer School in Logic, Language and Information (ESSLLI)**. Utrecht University (1999). http://folli.loria.fr/cds/1999/esslli99/courses/barr-wells.html.

[99] Benjamin Russell. Motivating Category Theory for Haskell for Non-mathematicians. http://dekudekuplex.wordpress.com/2009/01/19/motivating-learning-category-theory-for-non-mathematicians/.

[100] Arrow documentation. http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Arrow.html.

[101] Ross Paterson. Arrows: A general interface to computation. http://www.haskell.org/arrows/.

[102] John Hughes. Generalising monads to arrows. **Sci. Comput. Program.**, 37(1-3):pages 67–111 (2000).

[103] Sam Lindley, Philip Wadler, and Jeremy Yallop. The arrow calculus. **Submitted to ICFP** (2008). http://homepages.inf.ed.ac.uk/wadler/papers/arrows/arrows.pdf.

[104] Ross Paterson. Programming with Arrows. In Jeremy Gibbons and Oege de Moor (editors), **The Fun of Programming**, pages 201–222. "Cornerstones of Computing" series, Palgrave (March 2003). `http://www.soi.city.ac.uk/~ross/papers/fop.html`.

[105] Ross Paterson. A new notation for arrows. In **ICFP '01: Proceedings of the sixth ACM SIGPLAN international conference on Functional programming**, pages 229–240. ACM, New York, NY, USA (2001).

[106] John Hughes. Programming with Arrows. In Varmo Vene and Tarmo Uustalu (editors), **5th Internaltional Summer School on Advanced Functional Programming**, pages 73–129. Number 3622 in LNCS, Springer (2005).

[107] Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. In **Proceedings of the workshop on Mathematically Structured Functional Programming (MSFP)** (2008). `http://homepages.inf.ed.ac.uk/wadler/papers/arrows-and-idioms/arrows-and-idioms.pdf`.

[108] Antony Courtney, Paul Hudak, Henrik Nilsson, and John Peterson. Yampa: Functional Reactive Programming with Arrows. `Yampa:FunctionalReactiveProgrammingwithArrows`.

[109] Haskell XML Toolbox. `http://www.fh-wedel.de/~si/HXmlToolbox/`.

[110] Wolfgang Jeltsch. Grapefruit. `http://haskell.org/haskellwiki/Grapefruit`.

[111] Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer. There and back again: arrows for invertible programming. In **Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell**, pages 86–97. ACM, New York, NY, USA (2005).

[112] Dan Piponi. Evaluating cellular automata is comonadic. `http://blog.sigfpe.com/2006/12/evaluating-cellular-automata-is.html`.

[113] Conal Elliott. Functional interactive behavior. `http://conal.net/blog/posts/functional-interactive-behavior/`.

[114] Sterling Clover. Comonads in everyday life. `http://fmapfixreturn.wordpress.com/2008/07/09/comonads-in-everyday-life/`.

[115] Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. **Electron. Notes Theor. Comput. Sci.**, 203(5):pages 263–284 (2008).

[116] Tarmo Uustalu and Varmo Vene. The dual of substitution is redecoration. In **Trends in functional programming**, pages 99–110. Intellect Books, Exeter, UK (2002).

[117] Venanzio Capretta, Tarmo Uustalu, and Varmo Vene. Recursive coalgebras from comonads. **Inf. Comput.**, 204(4):pages 437–468 (2006).

[118] Tarmo Uustalu, Varmo Vene, and Alberto Pardo. Recursion schemes from comonads. **Nordic J. of Computing**, 8(3):pages 366–390 (2001).

[119] Tarmo Uustalu and Varmo Vene. The Essence of Dataflow Programming. In **Central European Functional Programming School**, pages 135–167 (2006).

[120] Brent Yorgey. blog :: Brent -> [String]. `http://byorgey.wordpress.com/`.

[121] Brent Yorgey. `http://www.cis.upenn.edu/~byorgey/`.

[122] University of Pennsylvania programming languages research group. `http://www.cis.upenn.edu/~plclub/`.

Book Review: "Real World Haskell"

by Chris Eidhof ⟨ce@tupil.com⟩
and Eelco Lempsink ⟨eml@tupil.com⟩

Real World Haskell [1] created quite a buzz both inside and outside the Haskell community. It has already received rave reviews, so we were eager to take a closer look for The Monad.Reader.

Details

Title	Real World Haskell
Authors	Bryan O'Sullivan, Don Stewart and John Goerzen
Pages	710
Published	November 2008
Publisher	O'Reilly Media
ISBN	0-596-51498-0 / 9780596514983

Content summary

Real World Haskell is a big book. With 28 chapters, two appendices, and over 700 pages it is certainly one of the most substantial Haskell books available. To give you some idea of the contents, we will give a short summary without explicitly enumerating every single chapter.

In the first four chapters, the basics of Haskell are introduced: creating datatypes, defining functions, working with GHCi and fundamental ideas of functional programming and how it compares to those languages with which most programmers are familiar. The target audience for this book is expected to know a thing or two about programming, and people familiar with the basics of Haskell can probably skip these chapters.

Then the real work starts. A library for representing and pretty printing JSON (JavaScript Object Notation) is written and the reader gets introduced to typeclasses (while reflecting on the JSON chapter to show how the code can be improved), including a comprehensive overview of the different numeric types, their features and how to convert between them.

I/O is introduced in Chapter 7, without mentioning 'the M word' until the end of the chapter. In the next three chapters, libraries are written for file processing and advanced filesystem searching (comparable to the Unix `find` command) where each chapter builds on the previous chapters.

Chapter 10 and 12 together are used to build a library to read a barcode from an image. Along the way the Maybe monad is introduced, without explicitly being mentioned. As an intermezzo, Chapter 11 shows how to do testing and quality assurance using QuickCheck and HPC.

Then there are several chapters about more advanced data structures, such as `Data.Sequence`, and of course about monads, monad transformers, and using Parsec. Monads are taught by looking at examples from the previous chapters. Some monad instances were already introduced (without calling them monads) and the chapter shows how to abstract over those examples.

Of course there is a chapter on interfacing with C, which is quite important for Real World programming. An interface to the Perl-compatible regular expression (PCRE) library is written in a straightforward way using `hsc2hs`.

There is a special chapter devoted to error handling, covering both pure and impure exceptions. As the book was written for GHC 6.8.3, the 'old' way of exception handling is used.

Chapter 20 to 23 show extended examples of very 'real world' programming such as systems programming (interacting with the filesystem, processes and using the `System.Time` module), working with databases, programming a web client (including XML parsing and saving to a database) and doing GUI programming (using gtk2hs). The chapters contain lots of code and relevant examples.

At the end of the book there are tips for speeding up your programs. There is a chapter on concurrent and multicore programming. A separate chapter covers profiling and optimization, which takes you through a lot of steps to optimize your programs. In the chapter about Building a Bloom Filter everything comes together: low-level code is transformed into a high-level library, there are bindings to C and QuickCheck properties, and the whole library is cabalized.

The penultimate chapter is about sockets and syslog (using low-level network communication). The last chapter introduces software transactional memory, which is used to build a concurrent web link checker that tries to find broken references in HTML files.

Review

It’s too bad *Real World Haskell* was not around when we started programming. This book is so full of interesting examples, tips, and advice that it is hard not to get enthusiastic about Haskell while reading it. Therefore we believe it will help foster a whole new generation of Haskell programmers.

The authors very clearly explain Haskell and the environment of the Haskell programmer. In most of the chapters, there is a good mix of code and text. The reader is taken through increasingly difficult examples step by step. New concepts and terminology are introduced along the way. Every chapter has well thought out examples.

Later in the book, most chapters focus on building one program or library and introduce a new concept along the way. One good example of this is the I/O case study in chapter 9: a library is built for searching the filesystem. The chapter is filled with little gems and builds a domain-specific language for searching the file system.

The book focuses on programmers coming from an imperative setting. The authors carefully relate Haskell to imperative languages, and give comparable code examples in languages like Python, Java and C, easing readers into doing functional programming. For example, they show how typeclasses relate to duck-typing. There are plenty of tips for avoiding possible imperative pitfalls.

The great thing about *Real World Haskell* is that they show how to solve real problems. The use of real problems to introduce concepts is a great way to teach: it motivates the reader to keep on learning and reading, because you get to build actual applications. The book moves away from a more theoretical way of teaching. In our opinion, this approach works so well that many other languages could benefit from a “real world” book like this.

Also, the book teaches more than just the Haskell language or the standard libraries: *Real World Haskell* gives an intuition for good programming style, helps the reader get acquainted with the Haskell environment and shows how to work with GHCi. For example: type errors are explained early on. While type errors might scare novice Haskell programmers, the authors give the best advice for dealing with them: don’t panic.

The book is also interesting for people that already know how to program in Haskell. It can give a new perspective on the language or serve as a reference book. Depending on your knowledge of Haskell, it might sometimes be a bit hard to find new things, but the elegant examples will make sure you will not get bored.

Since the release of the book, three months ago, a lot has already changed. For example, a new version of GHC has been released (the part about exceptions luckily contains a warning) and the HTTP library got a (much needed) update, to name a few. *Real World Haskell* is a great book, but it is not timeless. This

should not be a big issue though, because the concept of solving real problems itself will always work. We already look forward to the second edition.

It is inevitable that, in the process of writing such a large book so quickly, not all mistakes are caught. In the later chapters there are some code examples that are several pages with no text in between. While the code is well commented, it does not add much value to the book. Most of the *Real World Haskell* magic comes from the way that readers are guided through examples and incrementally shown how to grow a functional program.

Conclusion

Real World Haskell fulfills its promises and shows Haskell at full force tackling real problems. It is a delightful book to read and contains valuable information for many different levels of Haskell programmers. Even if you are an experienced Haskell programmer it is worthwhile to check out this book. You can most likely learn something from it yourself or, in the worst case, simply pass it on to your hacker friends that have not yet been properly introduced to functional programming.

In short, it is fair to say that *Real World Haskell* embiggens Haskell.

About The Reviewers

Chris Eidhof and Eelco Lempsink co-founded their company Tupil in 2008. They have a strong background in web programming, and are now exploring different ways to use functional programming as much as they can. They got introduced to Haskell at Utrecht University and fell in love with it at first sight. Chris and Eelco are both finishing their MSc at Utrecht.

References

[1] Bryan O'Sullivan, Donald Stewart, and John Goerzen. **Real World Haskell**. O'Reilly Media, Inc. (November 2008).

Calculating Monads with Category Theory

by Derek Elkins ⟨derek.a.elkins@gmail.com⟩

Several concepts in Haskell are inspired by or taken from category theory. The most notable example being monads [1]. When these concepts are taken from category theory, however, very little of the theory around them is brought along. This is why, despite claims to the contrary, one can be a perfectly competent Haskell programmer without knowing anything at all about category theory. But, if you are familiar with category theory, you can turn around and apply it to Haskell.

One interesting question is: where do the common fundamental monads of Haskell, such as *State*, *Reader*, *Writer*, *List*, and *Maybe*, come from? Historically, they were recognized from repeated patterns in the denotational semantics of computer programs. From this we would expect those monads to be rather ad-hoc from a categorical perspective. It turns out that some arise from constructs of fundamental importance in category theory, in particular, *State* and *Cont*. I'm not sure what the significance of this is, but I do know that this means it will be much easier to apply categorical results to such monads – and that there will be more results to apply.

One thing to note: the goal here is **not** to provide **intuition** about monads, but instead to provide tools for analyzing and building them and related structures. One could even say that the goal is how you could arrive at these monads **without** intuition. On the other hand, along the way a bit of intuition for the categorical concepts used will hopefully be built. The reader can consider unproven or partially proven statements to be exercises.

Categories

It is said that the inventor of category theory, Saunders Mac Lane, invented categories so he could talk about functors which he invented so he could talk about natural transformations [2]. The concept we will be making most use of is the concept of adjunctions which requires an understanding of natural transformations so we will need the same sequence of ideas.

So we will start with the basic definition of a category. For a more comprehensive introduction, I recommend Barr and Wells lecture notes [3] and/or Awodey's book [4]

Definition 1 (Category)**.** A **category** is a mathematical structure consisting of a set of "objects" and for each pair of objects, A and B, a set of "arrows," $\mathrm{Hom}(A, B)$. Objects and arrows can be anything. We can think of the objects as types that specify the source and target type for arrows. We also require for each object, A, an identity arrow $\mathrm{id} : A \to A$ and a composition operator, $\circ$, for arrows that is defined whenever it is well-typed. These should satisfy, for all arrows f, g, and h: $\mathrm{id} \circ f = f = f \circ \mathrm{id}$ and $f \circ (g \circ h) = (f \circ g) \circ h$.

Note that the Hom notation is implicitly parameterized on the category. Multiple uses of Hom may refer to potentially different categories.

The module *Control.Category* has the *Category* type class that captures this idea.

```
  -- From Control.Category
class Category hom where
  id  :: hom a a
  (∘) :: hom b c → hom a b → hom a c
```

Listing 47: *Category* type class

One obvious instance of the *Category* class is the $(\to)$ instance. Our main example of a category will be the category **Hask** of (idealized) Haskell types and functions which corresponds to this instance of *Category*.

Given any category we can form its **opposite category** by simply flipping the arrows around and composing in the opposite order. We'll have some use for the opposite category of **Hask** written $\mathbf{Hask}^{op}$. We encode it into Haskell with the newtype $\leftarrow$, defined in Listing 48.

With just this definition of category we can start making very general and useful ideas. One of the most prevalent ideas is the notion of **isomorphism**.

```
newtype (a ← b) = Op {unOp :: b → a}
instance Category (←) where
  id           = Op id
  Op f ∘ Op g = Op (g ∘ f)
```

Listing 48: Flipped → type

Definition 2 (Isomorphism)**.** An arrow $f : A \to B$ is an **isomorphism** if there exists an arrow $g : B \to A$ such that $f \circ g = \mathrm{id}$ and $g \circ f = \mathrm{id}$.

It is easy to prove that if f is an isomorphism then g is uniquely determined and so we will often use the notation f^{-1} for that g. We say two objects are isomorphic, written $A \cong B$ for objects A and B, if there exists an isomorphism between them. Objects can be isomorphic in many different ways and so, for example, an object can be non-trivially isomorphic to itself. Isomorphic objects are equivalent for all categorical purposes, and so often we'll say "the object" that satisfies some property when really there are potentially many isomorphic objects that satisfy it. Since calculating the inverse of a function is undecidable we'll represent isomorphisms by pairs of arrows that are assumed to be inverses. Every **newtype** is isomorphic to its underlying type and we will often make *Iso* values corresponding to these isomorphisms.

```
  -- Law: to iso . from iso = id and from iso . to iso = id
data GeneralizedIso arr a b = Iso {to :: arr a b, from :: arr b a}
type Iso = GeneralizedIso (→)

flipIso :: GeneralizedIso arr a b → GeneralizedIso arr b a
flipIso (Iso t f) = Iso f t

  -- An example of an isomorphism induced by a newtype.
  -- The rest of the examples are in the Appendix.
isoOp :: Iso (b → a) (a ← b)
isoOp = Iso Op unOp
```

Listing 49: A data type of isomorphisms

Functors

The next concept is functors, or mappings between categories.

Definition 3 (Functor)**.** A **functor** $F : \mathbf{C} \to \mathbf{D}$ between categories $\mathbf{C}$ and $\mathbf{D}$ is composed of an action on objects, $A \in \mathbf{C}$ being sent to $FA \in \mathbf{D}$, and an action on arrows taking an arrow $f : A \to B$ in $\mathbf{C}$ to an arrow $Ff : FA \to FB$ satisfying $F\mathrm{id} = \mathrm{id}$ and $F(f \circ g) = Ff \circ Fg$.

In Haskell instances of the *Functor* type class correspond to functors from **Hask** to **Hask**. The type constructor is the action on objects and *fmap* is the action on arrows. Unfortunately, we'll need functors between more than just **Hask** so we will generalize the *Functor* type class.

```
  -- Same laws as Functor:
  -- fmap' id = id and fmap' (f . g) = fmap' f . fmap' g
class Functor' arr arr' f where
  fmap' :: arr a b → arr' (f a) (f b)
instance (Functor f) ⇒ Functor' (→) (→) f where
  fmap' = fmap
instance (CoFunctor f) ⇒ Functor' (←) (→) f where
  fmap' = cofmap ∘ unOp
  -- Contravariant functor laws:
  -- cofmap id = id and cofmap (f . g) = cofmap g . cofmap f
class CoFunctor f where
  cofmap :: (a → b) → f b → f a
instance CoFunctor ((←) r) where
  cofmap t = Op ∘ (∘t) ∘ unOp
```

Listing 50: A generalized functor class and some supporting classes and instances

Here *arr* and *arr′* represent the source and target categories of the functor. The first instance corresponds to functors from **Hask** to **Hask** just like the normal *Functor* type class. The second instance corresponds to functors from $\mathbf{Hask}^{op}$ to **Hask** where *CoFunctor* represents contravariant functors which are simply normal functors from an opposite category, in this case $\mathbf{Hask}^{op}$. Instances of *Functor* are called covariant functors when there is a need to distinguish them. There are more supporting instances in the Appendix.

Functors compose in a natural way and there is an identity functor so we arrive at **Cat**, the category of categories and functors between them. When rendering this into Haskell we add a phantom type parameter *arr* to specify the intermediate category to give something for the type class mechanism to dispatch on.

```
  -- Functor composition
newtype (f ∘ g) arr x = O {unO :: f (g x)}
instance (Functor f, Functor g) ⇒ Functor ((f ∘ g) (→)) where
  fmap t = O ∘ fmap (fmap t) ∘ unO
instance (CoFunctor f, CoFunctor g) ⇒ Functor ((f ∘ g) (←)) where
  fmap t = O ∘ cofmap (cofmap t) ∘ unO
newtype Id a = Id {unId :: a}
instance Functor Id where
  fmap f = Id ∘ f ∘ unId
```

Listing 51: Identity and composition of functors

Two extremely important examples of functors are the covariant and contravariant Hom-functors. Given an arrow $f : A \to B$ we have $\mathrm{Hom}(f, C) : \mathrm{Hom}(B, C) \to \mathrm{Hom}(A, C)$ and $\mathrm{Hom}(C, f) : \mathrm{Hom}(C, A) \to \mathrm{Hom}(C, B)$. So Hom is contravariant in its first argument and covariant in its second argument. We'll use a special notation for the Hom-functors that makes their behavior quite clear. We'll write $\mathrm{Hom}(f, C)$ as $(\circ f)$ and $\mathrm{Hom}(C, f)$ as $(f \circ)$. The former case corresponds to the *CoFunctor* instance for $(\leftarrow)$ r and the latter case *Functor* instance for $(\rightarrow)$ r from *Control.Monad.Instances*.

Natural Transformations

The final basic definition we need before talking about adjunctions is natural transformations. Given functors F and G between the same category, we can view objects and arrows of the form FA and Ff as "structured" in some way represented by F. One would like to be able to talk about transformations between such "structures" generically. Natural transformations allow us to do this, "natural" meaning uniform.

Definition 4. Given functors $F, G : \mathbf{C} \to \mathbf{D}$, a **natural transformation** $\alpha : F \to G$ is a family of arrows $\alpha_A : FA \to GA$, called **components**, for each object A in $\mathbf{C}$

such that for any arrow $f : A \to B$ the following equation, called the **naturality condition**, holds: $Gf \circ \alpha_A = \alpha_B \circ Ff$.

Often it will be convenient to state properties of such natural transformations component-wise, e.g. $FA \cong GA$, in this case such statements are said to be **natural in** the relevant components. So for the example, one would say, that $FA \cong GA$ is natural in A and this would mean the natural isomorphism (i.e. just an isomorphism in a functor category) $F \cong G$.

For a Haskell programmer one Haskell concept should immediately jump to mind. A natural transformation is a polymorphic function and the naturality condition is the free theorem [5]. This can be captured as a type synonym, but not one we'll have much use for.

```
  -- f and g functors between the same categories
type Nat arr f g = ∀ a.arr (f a) (g a)
```

Listing 52: Natural transformations

Natural transformations compose component-wise and there is an identity natural transformation that just has identity arrows as components, so we have another example of a category, or rather a whole family of them. Given any two categories $\mathbf{C}$ and $\mathbf{D}$ we have the **functor category**, written $[\mathbf{C}, \mathbf{D}]$, whose objects are functors and arrows are natural transformations. The component-wise composition is called **vertical composition**. We'll write the Hom-sets for $[\mathbf{C}, \mathbf{D}]$, $\mathrm{Hom}(F, G)$ as $\mathrm{Nat}(F, G)$ for emphasis. This "arrows between arrows" situation is called a **2-category** with $\mathbf{Cat}$ being the archetypical example. There is another way of composing natural transformations called **horizontal composition**. Given two natural transformations $\alpha : F \to G$ and $\beta : H \to K$, the horizontal composition is $\alpha \star \beta : F \circ H \to G \circ K$. We'll only ever use two special cases of horizontal composition. Firstly, we will use $\mathrm{id}_F \star \beta$, which we will write as $F\beta$. This corresponds to the natural transformation with components $F\beta_A$ or in Haskell notation *fmap* β. The second case is $\alpha \star \mathrm{id}_H$, which we will write as α_H and means the natural transformation with components α_{HA} which in Haskell has no particular notation – since type application is implicit – and corresponds simply to restricting the polymorphic function α to types of the form HA. Due to the **interchange law**, $(\alpha \circ \beta) \star (\gamma \circ \delta) = (\alpha \star \gamma) \circ (\beta \star \gamma)$, we can define $\alpha \star \beta$ in terms of the special cases, i.e. $\alpha_H \circ F\beta = \alpha \star \beta = F\beta \circ \alpha_H$.

One very important, but simple, result in category theory is the **Yoneda Lemma**. All we need is a (slightly) special case of it.

Lemma 5 (Yoneda)**.**

$$\mathrm{Nat}(\lambda C.\mathrm{Hom}(B, C), \lambda C.\mathrm{Hom}(A, C)) \quad \cong \quad \mathrm{Hom}(A, B)$$

natural in A *and* B*.*

```
yoneda :: ∀ arr x y.(Category arr)
        ⇒ GeneralizedIso (→) (∀ a.arr a x → arr a y) (arr x y)
yoneda = Iso to' from'
  where to' :: (∀ a.arr a x → arr a y) → arr x y
        to' alpha = alpha id
        from' :: arr x y → ∀ a.arr a x → arr a y
        from' f = (f∘)
```

Listing 53: The Yoneda lemma rendered in Haskell

Another way of stating this result is by introducing the Yoneda functor defined by $\mathcal{Y} = \lambda X.\lambda Y.\mathrm{Hom}(X, Y)$. The Yoneda lemma now states that for all A and B, $\mathrm{Hom}(A, B) \cong \mathrm{Hom}(\mathcal{Y}A, \mathcal{Y}B)$ which is the very definition of $\mathcal{Y}$ being fully, faithful. In general, a fully, faithful functor reflects isomorphisms, that is if F is a full and faithful functor and $FA \cong FB$ then $A \cong B$. This particular facet of the Yoneda is used **constantly**.

```
reflectIso :: (Category arr, Category arr', Functor' arr' arr f)
           ⇒ (∀ a b.Iso (arr (f a) (f b)) (arr' a b))
           → GeneralizedIso arr (f a) (f b)
           → GeneralizedIso arr' a b
reflectIso iso (Iso to2 from2) = Iso (to iso to2) (to iso from2)
```

Listing 54: Fully, faithful functors reflect isomorphisms

Adjunctions

Recapping, so far we have seen categories and, in particular, the category **Hask**. We have defined functors which are arrows between categories which correspond to

type constructors that are an instance of *Functor′*. Finally, we have seen natural transformations which are arrows between functors and correspond to polymorphic functions. With these basic definitions we can define one of the fundamental constructions of category theory: adjunctions. There are a few different ways of presenting adjunctions but the most compact and, in my opinion, most intuitive and most useful for calculation is the following.

Definition 6 (Adjunction)**.** A functor $F : \mathbf{C} \to \mathbf{D}$ is **left adjoint** to $U : \mathbf{D} \to \mathbf{C}$, written $F \dashv U$, if there exists an isomorphism $\mathrm{Hom}(FA, B) \cong \mathrm{Hom}(A, UB)$ natural in A and B.

U is called the **right adjoint** of F. We can render this quite simply in Haskell.

```
type GeneralAdjunction arr arr' f u
  = ∀ a b.Iso (arr (f a) b) (arr' a (u b))
type Adjunction f u = GeneralAdjunction (→) (→) f u
```

Listing 55: Adjunctions

An important and familiar example is the adjunction defined by curry/uncurry showing that $\lambda A.(A, B) \dashv \lambda A.B \to A$.

```
stateAdjunction :: Adjunction ((,) s) ((→) s)
stateAdjunction = Iso (flip ∘ curry) (uncurry ∘ flip)
```

Listing 56: The adjunction induced by curry and uncurry, the naming will become clear later

For warm-up, let's prove that if $F \dashv U$ and $F \dashv U'$ then $U \cong U'$ i.e. right adjoints are unique up to isomorphism. By the definition of adjunction we have $\mathrm{Hom}(A, UB) \cong \mathrm{Hom}(FA, B) \cong \mathrm{Hom}(A, U'B)$ and so, by transitivity, we have $\mathrm{Hom}(A, UB) \cong \mathrm{Hom}(A, UB)$ and thus Yoneda gives us $U \cong U'$. The succinctness and simplicity of this proof is what I mean by this presentation being good for calculation.

Let's use $\phi_{A,B}$ to name the isomorphism between $\mathrm{Hom}(FA, B)$ and $\mathrm{Hom}(A, UB)$. We can define what are called the unit and counit of the adjunction as follows: The unit, traditionally written η, is defined as $\eta_A = \phi_{A,FA}(\mathrm{id})$. Similarly, the counit

is $\varepsilon_B = \phi^{-1}_{UB,B}(\mathrm{id})$. For our curry/uncurry adjunction the counit is the function $\lambda(f,x).fx$ i.e. it is application which has obvious importance. The unit is the function $\lambda a.\lambda b.(a,b)$, a function that doesn't seem particularly remarkable. What happens when we apply the $(B \to)$ functor's action to it, i.e. *fmap f* $\circ$ *unit*? Well this is the function $\lambda a.\lambda b.f(a,b)$ or just *curry f*. This is a general property of units, namely $\phi(f) = Ff \circ \eta$ and similarly $\phi^{-1}(f) = \varepsilon \circ Uf$. Certainly this is consistent with $\eta = \phi(\mathrm{id}) = F\mathrm{id} \circ \eta = \mathrm{id} \circ \eta = \eta$. Thus, a second presentation of adjunctions: $F \dashv U$ if there exists natural transformations $\eta : \mathrm{Id} \to U \circ F$ and $\varepsilon : F \circ U \to \mathrm{Id}$ that satisfy the **triangle identites** $U\varepsilon \circ \eta_U = \mathrm{id}_U$ and $\varepsilon_F \circ F\eta = \mathrm{id}_F$. I'll leave it as an exercise to show that the definitions $\eta = \phi(\mathrm{id})$ and $\varepsilon = \phi^{-1}(\mathrm{id})$ imply that η and ε satisfy the triangle identities and that the triangle identities are exactly what are required to make $\phi(f) = Ff \circ \eta$ into a natural isomorphism.

```
unit :: (Category arr, Category arr', FromIso arr')
     => GeneralAdjunction arr arr' f u -> arr' a ((u . f) arr a)
unit adj = fromIso isoO . to adj id
counit :: (Category arr, Category arr', FromIso arr)
       => GeneralAdjunction arr arr' f u -> arr ((f . u) arr' b) b
counit adj = from adj id . fromIso flipIsoO
```

Listing 57: Units and counits

Monads and Comonads

Okay. So what does this have to do ith Haskell and monads? Well, the first result is that every adjunction gives rise to a monad and a comonad. As a reminder the usual categorical presentation of a monad is a functor $T : \mathbf{C} \to \mathbf{C}$ and two natural transformations $\eta : \mathrm{Id} \to T$ and $\mu : T \circ T \to T$ correspond to *return* and *join* respectively in Haskell. Dually, a comonad is a functor $G : \mathbf{C} \to \mathbf{C}$ with two natural transformations $\varepsilon : G \to \mathrm{Id}$ and $\delta : G \to G \circ G$ often called *extract* and *duplicate* in Haskell. As the names suggests, η and ε are the unit and counit of an adjunction and so we can see that $T = U \circ F$ and $G = F \circ U$ form an adjunction $F \dashv U$. The definition of μ and δ in terms of what we have in an adjunction can be found simply by trying to make the types fit. We end up with $\mu = U\varepsilon_F$ and $\delta = F\eta_U$. It's a relatively straightforward exercise to show that these definitions do satisfy the monad and comonad laws.

```
multiplication :: ∀ arr arr' f u a.(Category arr, Category arr',
                                    FromIso arr, FromIso arr',
                                    Functor' arr arr' u,
                                    Functor' arr' arr' ((u ∘ f) arr))
                 ⇒ GeneralAdjunction arr arr' f u
                 → arr' ((u ∘ f) arr ((u ∘ f) arr a)) ((u ∘ f) arr a)
multiplication adj =
  fromIso isoO
   ∘ fmap' (counit adj ∘ fromIso isoO)
   ∘ fromIso flipIsoO
   ∘ fmap' (fromIso flipIsoO :: arr' ((u ∘ f) arr a) (u (f a)))
comultiplication :: ∀ arr arr' f u a.(Category arr, Category arr',
                                      FromIso arr, FromIso arr',
                                      Functor' arr' arr f,
                                      Functor' arr arr ((f ∘ u) arr'))
                   ⇒ GeneralAdjunction arr arr' f u
                   → arr ((f ∘ u) arr' a) ((f ∘ u) arr' ((f ∘ u) arr' a))
comultiplication adj =
  fmap' (fromIso isoO :: arr (f (u a)) ((f ∘ u) arr' a))
   ∘ fromIso isoO
   ∘ fmap' (fromIso flipIsoO ∘ unit adj)
   ∘ fromIso flipIsoO
```

Listing 58: Monad "multiplication" and comonad "comultiplication", i.e. μ and δ

Using our previous adjunction example it is not hard to convince one's self that the monad it gives rise to is the *State* monad. The comonad that it gives rise to has been called the State-in-Context comonad [6]. Another example of an adjunction, $\mathrm{Id} \dashv \mathrm{Id}$, gives rise to the *Identity* (co)monad. This trivial adjunction will turn out to be more useful than it first appears. Another significant adjunction is the one induced by *flip* namely $\lambda A.(A \to R)^{op} \dashv \lambda A.(A \to R)$. This one is different than the examples we've had before. If we look at the type of *flip* we get $\mathrm{Hom}(A, B \to C) \cong \mathrm{Hom}(B, A \to C)$ which does not look like the right form for an adjunction. The key here is one side can be viewed as being in the opposite category. This situation is described as $\lambda A.(A \to R)$ being self-adjoint to itself on the right. The monad induced by this adjunction looks like and indeed is the continuation monad. It turns out, in this case, that the comonad is the same thing. Here is where the generalized *Functor′* class will come in handy.

That gives us three monads, two of which are rather significant. What about some of the others? *Reader* and *Writer* come to mind as they seem to have some relation to the adjunction underlying the *State* monad. This becomes more striking when we consider comonads as well. For example, there is the *Writer* monad which is $\lambda A.(A, M)$ where M is a monoid, and there is also a comonad $\lambda A.M \to A$ where M is also a monoid. Does the *State* adjunction have anything to say about this relationship? As it turns out, given an adjunction $F \dashv U$ if and only if when F is a comonad, U is a monad. A similar statement also holds if F is a monad and U is a comonad. We could again beat pieces together until they type check and check if the required properties hold, but in this case there is a much more elegant way. If F is a comonad then we have a natural transformation $\varepsilon : F \to \mathrm{Id}$ and we'd expect this to somehow induce the natural transformation $\eta : \mathrm{Id} \to U$. Let's see if we can solve a more general problem, that of inducing a natural transformation $\beta : U' \to U$ given $\alpha : F \to F'$, $F \dashv U$, and $F' \dashv U'$. Using the definition of adjunction, we can see that it is at least sensible: $\mathrm{Hom}(\alpha_A, B) : \mathrm{Hom}(F'A, B) \to \mathrm{Hom}(FA, B)$ and $\mathrm{Hom}(A, \beta_B) : \mathrm{Hom}(A, U'B) \to \mathrm{Hom}(A, UB)$, no, the first case is not a typo, Hom is contravariant in it's first argument so the order gets swapped. This suggests a notion which is called a **transformation of an adjunction** from $F' \dashv U'$ to $F \dashv U$, we just need to require that it takes the defining isomorphism of hom-sets to another isomorphism of hom-sets, i.e. if $\phi'_{A,B} : \mathrm{Hom}(F'A, B) \cong \mathrm{Hom}(A, U'B)$ and $\phi_{A,B} : \mathrm{Hom}(FA, B) \cong \mathrm{Hom}(A, UB)$ then we require $\phi \circ (\circ\alpha) = (\beta\circ) \circ \phi'$, the motivation for the name "transformation" is clear. By Yoneda, if we have α then there is only one β that will lead to a transformation of adjunctions. This β is called the **conjugate** of α and similarly α is the conjugate of β.

Now the Id adjunction comes in handy allowing us to transport units/counits across an adjunction. To transport the (co)multiplication of the (co)monads we need to show that if $F \dashv U$ then $F \circ F \dashv U \circ U$. We'll show more generally that you can compose two adjunctions $F \dashv U$ and $H \dashv K$ giving $F \circ H \dashv K \circ U$. With

```
type State s = ((→) s ∘ (,) s) (→)
type CoState s = ((,) s ∘ (→) s) (→)

bind :: (Functor t)
     ⇒ (∀ a.t (t a) → t a)
     → (∀ a.t a → (a → t b) → t b)
bind join = flip (λk → join ∘ fmap k)

instance Monad (State s) where
  return = unit stateAdjunction
  (>>=)  = bind (multiplication stateAdjunction)

instance Comonad (CoState s) where
  extract   = counit stateAdjunction
  duplicate = comultiplication stateAdjunction

idAdjunction :: (Category arr, FromIso arr)
             ⇒ GeneralAdjunction arr arr Id Id
idAdjunction = Iso ((fromIso isoId∘) ∘ (∘fromIso isoId))
                   ((fromIso flipIsoId∘) ∘ (∘fromIso flipIsoId))

contAdjunction :: GeneralAdjunction (←) (→) ((←) r) ((←) r)
contAdjunction = Iso ((Op∘) ∘ flip ∘ (unOp∘) ∘ unOp)
                     (Op ∘ (Op∘) ∘ flip ∘ (unOp∘))

type Cont r = ((←) r ∘ (←) r) (←)

instance Monad (Cont r) where
  return = unit contAdjunction
  (>>=)  = bind (multiplication contAdjunction)
```

Listing 59: Monads and comonads from adjunctions

```
rightConjugate :: (Category arr, Category arr')
               => GeneralAdjunction arr arr' f u
               -> GeneralAdjunction arr arr' f' u'
               -> Nat arr f f'
               -> Nat arr' u' u
rightConjugate adj1 adj2 sigma = (to adj1 ∘ (∘sigma) ∘ from adj2) id
leftConjugate :: (Category arr, Category arr')
              => GeneralAdjunction arr arr' f u
              -> GeneralAdjunction arr arr' f' u'
              -> Nat arr' u' u
              -> Nat arr f f'
leftConjugate adj1 adj2 tau = (from adj1 ∘ (tau∘) ∘ to adj2) id
```

Listing 60: Conjugates

those we can transport monads and comonads across an adjunction.

The ability to compose adjunctions plus the identity adjunction suggests that adjunctions are arrows in a category. This is the case: the objects are categories and an adjunction $F \dashv U$ is an arrow from $\mathbf{C}$ to $\mathbf{D}$ if $U : \mathbf{C} \to \mathbf{D}$. Transformations of adjunctions are then arrows between these arrows so this is another example of a 2-category like **Cat**. Adjunctions compose nicely unlike monads [7] or comonads so we can combine the adjunctions underlying two monads to get a new combined monad. Currently, we have only two non-trivial adjunctions: the one underlying *State* which is **Hask** $\to$ **Hask** and the one underlying *Cont* which is $\mathbf{Hask}^{op} \to$ **Hask**. We can compose the *State* one with itself any number of times and with different parameterizations leading to a *State* monad with multiple pieces and types of state, but these are all isomorphic to a *State* monad with a tuple for the state. The other possibility is to compose the *Cont* adjunction and the *State* adjunction giving the continuation state monad.

Conclusion

The purpose of this article was to show how Haskell can take more than just definitions from category theory. Haskell can also take the theorems and results in the form of executable code. The central concept adjunctions was used. Even though adjunctions are not commonly mentioned in the context of functional programming they are extremely important in category theory. There are other examples

```
composeAdjunction ::
  (Category arr, Category arr', Category arr'',
  FromIso arr, FromIso arr'')
  ⇒ GeneralAdjunction arr arr' f u
  → GeneralAdjunction arr' arr'' f' u'
  → GeneralAdjunction arr arr'' ((f ∘ f') arr') ((u' ∘ u) arr')
composeAdjunction adj1 adj2
  = Iso ((fromIso isoO∘) ∘ to adj2 ∘ to adj1 ∘ (∘fromIso isoO))
        ((∘fromIso flipIsoO) ∘ from adj1 ∘ from adj2 ∘ (fromIso flipIsoO∘))
doubleAdjunction :: (Category arr, FromIso arr)
                 ⇒ GeneralAdjunction arr arr f u
                 → GeneralAdjunction arr arr ((f ∘ f) arr) ((u ∘ u) arr)
doubleAdjunction adj = composeAdjunction adj adj
```

Listing 61: Composition of adjunctions

of category theory being directly used as executable code. One of them is free monads [8], which cover the cases of *Maybe* and *Either*. Generic folds are another example [9]. The **category-extras** package [10] describes many categorical ideas specialized to **Hask** and closely related categories.

Another way of using category theory to produce Haskell code is to work things out with pencil and paper first and only render the final result in Haskell. For example, there's a functor that takes a monoid and forgets the operation and unit giving only a set which is called the underlying set functor. This functor has a left adjoint called the free monoid functor. A free monoid is just another term for a list and the monad induced by this adjunction is the list monad. Another example arises from the view of monads as monoid objects in the category of endofunctors. A category can be viewed as a generalized monoid, in particular, as a "typed" monoid, and we can talk about category objects in a category. This leads to the idea of talking about category objects in the category of endofunctors and arriving at a notion rather like the "indexed" or "parameterized" monads that have been suggested over the years [11]. While these arguments could be rendered into Haskell imperfectly, they would require machinations and duplications of the tools used above. Haskell's type system just isn't capable of capturing the uniformities or, in many cases, even representing the structures needed, so trying to encode them is quite painful and often impossible even if the final result of these constructions is readily translatable to Haskell.

```
conjugateComonadExtract :: (Monad f)
                        ⇒ Adjunction f u → u b → b
conjugateComonadExtract adj
  = unId ∘ rightConjugate idAdjunction adj (return ∘ unId)
conjugateMonadReturn :: (Comonad u)
                     ⇒ Adjunction f u → a → f a
conjugateMonadReturn adj
  = leftConjugate idAdjunction adj (Id ∘ extract) ∘ Id
conjugateComonadExtract' :: (Monad u)
                         ⇒ Adjunction f u → f a → a
conjugateComonadExtract' adj
  = unId ∘ leftConjugate adj idAdjunction (return ∘ unId)
conjugateMonadReturn' :: (Comonad f)
                      ⇒ Adjunction f u → b → u b
conjugateMonadReturn' adj
  = rightConjugate adj idAdjunction (Id ∘ extract) ∘ Id
conjugateComonadDuplicate :: (Monad f)
                          ⇒ Adjunction f u → u b → u (u b)
conjugateComonadDuplicate adj
  = unO ∘ rightConjugate (doubleAdjunction adj) adj (join ∘ unO)
conjugateMonadJoin :: (Comonad u)
                   ⇒ Adjunction f u → f (f a) → f a
conjugateMonadJoin adj
  = leftConjugate (doubleAdjunction adj) adj (O ∘ duplicate) ∘ O
conjugateComonadDuplicate' :: (Monad u)
                           ⇒ Adjunction f u → f a → f (f a)
conjugateComonadDuplicate' adj
  = unO ∘ leftConjugate adj (doubleAdjunction adj) (join ∘ unO)
conjugateMonadJoin' :: (Comonad f)
                    ⇒ Adjunction f u → u (u b) → u b
conjugateMonadJoin' adj
  = rightConjugate adj (doubleAdjunction adj) (O ∘ duplicate) ∘ O
```

Listing 62: Conjugating monads and comonads across adjunctions

```
  -- This uses the ((->) r) Monad instance from Control.Monad.Instances
instance Comonad ((,) r) where
  extract   = conjugateComonadExtract' stateAdjunction
  duplicate = conjugateComonadDuplicate' stateAdjunction

instance Monoid r => Comonad ((->) r) where
  extract     = ($mempty)
  duplicate f = \r1 r2 -> f (r1 `mappend` r2)

  -- This is using the above Comonad instance
instance Monoid w => Monad ((,) w) where
  return = conjugateMonadReturn stateAdjunction
  (>>=)  = bind (conjugateMonadJoin stateAdjunction)
```

Listing 63: Examples of conjugating monads and comonads

```
contStateAdjunction :: GeneralAdjunction (<-) (->) (((<-) r . (,) s) (->))
                                                    (((->) s . (<-) r) (->))
contStateAdjunction = composeAdjunction contAdjunction stateAdjunction

type ContState r s = (((->) s . (<-) r) (->)
                      . ((<-) r . (,) s) (->)) (<-)

instance Monad (ContState r s) where
  return = unit contStateAdjunction
  (>>=)  = bind (multiplication contStateAdjunction)
```

Listing 64: The continuation state monad by composing adjunctions

While the definitions of monads, functors, and folds have been used in Haskell, very little of the theory surrounding them has followed. Category theory still has plenty to offer to Haskell and there is plenty of low-hanging fruit to be had.

About the author

Derek Elkins is an active-duty Airman in the United States Air Force stationed in west Texas. His interests include programming language theory, signal processing, and various areas of mathematics.

References

[1] Philip Wadler. Monads for functional programming. In **Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text**, pages 24–52. Springer-Verlag, London, UK (1995). `http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf`.

[2] Saunders M. Lane. **Categories for the Working Mathematician**. Springer-Verlag, New York (1998).

[3] Michael Barr and Charles Wells. Category theory - lecture notes for ESSLLI. page 123 (1999). `http://folli.loria.fr/cds/1999/library/pdf/barrwells.pdf`.

[4] Steve Awodey. **Category Theory**. Oxford University Press, USA (2006).

[5] Philip Wadler. Theorems for free! In **FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture**, pages 347–359. ACM, New York, NY, USA (1989). `http://homepages.inf.ed.ac.uk/wadler/papers/free/free.ps`.

[6] Richard B. Kieburtz. Codata and comonads in haskell (1999). `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.5169`.

[7] Christoph Lüth and Neil Ghani. Composing monads using coproducts. **SIGPLAN Not.**, 37(9):pages 133–144 (2002). `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.8.3581`.

[8] Wouter Swierstra. Data types à la carte. **Journal of Functional Programming**, 18(4):pages 423–436 (2008). `http://www.cse.chalmers.se/~wouter/Publications/DataTypesALaCarte.pdf`.

[9] Varmo Vene. **Categorical programming with inductive and coinductive types**. Ph.D. thesis, Faculty of Mathematics, University of Tartu (August 2000). `http://www.cs.ut.ee/~varmo/papers/`.

[10] http://hackage.haskell.org/cgi-bin/hackage-scripts/package/category-extras.

[11] Beyond monads. http://blog.sigfpe.com/2009/02/beyond-monads.html.

Appendix

```
isoO :: Iso (f (g x)) ((f ∘ g) arr x)
isoO = Iso O unO

flipIsoO :: Iso ((f ∘ g) arr x) (f (g x))
flipIsoO = flipIso isoO

isoId :: Iso x (Id x)
isoId = Iso Id unId

flipIsoId :: Iso (Id x) x
flipIsoId = flipIso isoId

class FromIso arr where
  fromIso :: Iso a b → arr a b

instance FromIso (→) where
  fromIso = to

instance FromIso (←) where
  fromIso = Op ∘ from
```

Listing 65: Support code for isomorphisms

```
instance (Functor f) ⇒ Functor' (←) (←) f where
  fmap' = Op ∘ fmap ∘ unOp
instance (CoFunctor f) ⇒ Functor' (→) (←) f where
  fmap' = Op ∘ cofmap
instance Functor f ⇒ CoFunctor ((f ∘ (←) r) (→)) where
  cofmap t = O ∘ fmap (cofmap t) ∘ unO
instance Functor f ⇒ CoFunctor (((←) r ∘ f) (→)) where
  cofmap t = O ∘ cofmap (fmap t) ∘ unO
class (Functor g) ⇒ Comonad g where
  extract   :: g a → a
  duplicate :: g a → g (g a)
instance Monad Id where
  return      = Id
  Id x >>= f = f x
instance Comonad Id where
  extract   = unId
  duplicate = Id
```

Listing 66: More support code for *Functor'*