

Teaching Statement

Richard A. Eisenberg

Computer science strikes me as beautiful: how the rules of logic can be used to prove the undecidability of the halting problem, how the magic wand of induction lets us use the solution of a problem to find the solution of that same problem, or how just a few typing rules can be the basis for all of known mathematics. Through teaching, I am able to share this beauty. At times, it can be frustrating. I can see the wondrous connections between, say, the mathematical induction a student learns in a number theory class and the recursive function on lists she just wrote, but she cannot. But then, teaching can be exhilarating; she gets it. Watching the spark in a student's eyes as she goes from a question to an answer thrills me every time.

My deep, abiding interest in education shows itself in many forms. I dedicated eight years of my life to sharing the beauty of computer science, teaching high school students introductory programming (Java) and AP Computer Science. At Penn, I was the sole instructor for two undergraduate courses, far surpassing the university's teaching requirement for PhD students. This past summer, I volunteered my time to organize and run a programming camp for middle- and high-school students. My very entry into the subject of my research—dependent types—was motivated by my desire to use the power of dependent types in my teaching.

I believe that my success in teaching stems from my indelible belief that every student truly *can* learn computer science. My expectations are always set high. I structure learning experiences to challenge my students and to force them to grow. When a student graduates from my class, I not only want them to understand and apply the subject matter on the syllabus, but I want to increase their confidence and strengthen their ability to work independently. A student's confidence is built when he masters a new skill not because the teacher taught the skill masterfully, but because the student himself sweated in the process. To encourage this growth, my assignments frequently require students to learn a topic independently and to apply their newfound knowledge. Doing this, students learn not only more subject matter, but also how to learn on their own, an invaluable skill as they enter the workplace. My students tend to find such tasks hard, but my experience is that students enroll in my classes because of this challenge, not in spite of it.

A teacher also can never forget the students' need for practical knowledge. It is not enough for a student to be able to say, after a course, "I now know how to program in Haskell." That student needs to be able to open a Haskell buffer and solve the data analysis problem that arose in her physics lab. Or he needs to be able to help debug his friend's effort at building a web app in Haskell. To this end, many of my assignments involve tackling real-world problems.

By way of example, I have included a copy of an assignment I used when I was the instructor for Penn's undergraduate Haskell course in 2014. The assignment asks students to use Haskell's *Monoid* type class and the `aeson` library to perform queries over a JSON dataset about locations of farmer's markets. During the assignment, students practice using I/O, learn more about the flexibility offered by Haskell's type classes, and must teach themselves how to use `aeson`, the state-of-the-art JSON parsing library professionals use. In so doing, students must both learn independently and complete a practical task. At the end of the semester, students highlighted this assignment, which I designed from scratch, as one of the most meaningful to them.

My classroom is a dynamic, welcoming place where students ask questions of me and of each other. As much as possible, I avoid teaching with static slides, preferring live coding. Students guide

me in writing the answer to a programming puzzle. Along the way, we get to see why several dead ends are indeed wrong. Teaching via live coding also exposes students to my thought process in writing code. When the lesson at hand does not require programming, I find that working with a chalkboard leads to a more intimate and fresh experience than slides would. The flexibility of building my figures as we go allows a question from a student to push the discussion in a new direction, and it allows me to tailor the lesson on the fly.

A key part of my classes is the opportunity for students to learn from their peers. When I teach a new subject in class, each individual student might learn only, say, 75% of the material. I often ask students to work on short exercises in pairs, a practice that allows the students to fill in each other's gaps. This tried-and-true technique benefits everyone: one student gets his gaps filled in, the other strengthens her understanding through teaching, and we all get our blood moving a bit. Working with peers also underscores my goal of learning without a teacher's direct intervention.

I take pride in my relationships with students outside of class. My office hours tend to be busy, and I invite students to show up, even if they don't have a specific question. I have seen students spontaneously form new partnerships and revel in solving a problem. I also have had the honor on many occasions to gain the trust of a student and to help see him or her through a thorny patch, whether in my class or in their life outside computer science. I have found, in the relationships that I've built with students over the years, that my enthusiasm for computer science is indeed infectious. Quite a few of my former students publicly credit my tutelage with helping them to decide to focus their education in college on computer science.

I have also enjoyed the opportunity to co-advise undergraduate research. One very successful project was with an undergraduate at Penn who wanted to hack on the Glasgow Haskell Compiler (GHC). As a core contributor to GHC, I became a co-advisor to his research project, an analysis and implementation of explicit type application in Haskell. The work resulted in a paper, currently in submission to the European Symposium on Programming (ESOP), and the feature that we designed will be a part of the next major release of GHC. This is an experience I hope to repeat as a college professor: work with a student to design a language feature, study the theory behind it, develop the code together, and then release. It is my hope that the ability to impact a language used both in academia and in industry will be enticing to undergraduate students and will encourage them to do research with me.

My considerable experience in teaching and the enjoyment I always find in a classroom get me excited about the next step: becoming a professor. In that role, I will be able to teach courses in programming languages, functional programming, compilers, and core computer science theory. I would also relish getting involved with teaching in the introductory sequence; I have always found helping students with their first steps in computer science to be richly rewarding. Lastly, I could easily refresh my knowledge to teach systems, architecture, or discrete math courses. I look forward to spending more time in the classroom and continuing to make a difference in the lives of my students and advisees.

CIS 194: Homework 6

Due Friday, October 17, 2014

No template file is provided for this homework. Download the `markets.json` file from the website, and make your `HW06.hs` Haskell file with your name, any sources you consulted, and any other relevant comments (just like in previous assignments). Then, say

```
{-# LANGUAGE DeriveGeneric, OverloadedStrings #-}
```

```
module HW06 where
```

```
import Data.Aeson
import Data.Monoid
import GHC.Generics
```

```
import qualified Data.ByteString.Lazy.Char8 as B
import qualified Data.Text                  as T
import qualified Data.Text.IO              as T
```

and off you go. Do make sure to include this module header, as it makes grading much easier for us.

Preface

Setup

You will need two packages that are not part of Haskell's standard library for this assignment. They are `aeson` and `text`. You can install these with `cabal update; cabal install aeson text`.¹ If you have `GHCi` open, you will need to restart `GHCi` to use these downloaded libraries.

¹ The `cabal update` part is to make sure you download the most recent versions of these packages.

Generics

You will see the language extension `DeriveGeneric` in the module header given above. This allows you to name the class `Generic`² in a deriving clause when declaring a new datatype. You will not use any of the methods or other features of the `Generic` class.³ The reason to derive `Generic` is for easy interoperability with the `aeson` JSON-parsing library. A derived `Generic` instance encodes various features of a datatype (such as its constructor names, any record-field names, etc.) that advanced Haskellers can (such as the authors of `aeson`) use to make your life easier.

² imported from `GHC.Generics`

³ Feel free to look it up!

JSON files

This homework centers around parsing and then querying information stored in a JSON file. JSON is a standardized data interchange format that is easy to read and easy to write. See json.org for the details, but you won't need to know about details for this assignment. Instead, the `aeson` library does all the work for you!

What you do have to worry about is making sure that your Haskell program can find your `markets.json` file. Putting the file in the same directory as your `HW06.hs` file is a great start, but it's not always enough. If you're having trouble getting your code to find your file, and you're using GHCi, try running `:!pwd`. That will print out the current directory GHCi thinks it's in. (The `:!` prefix allows you to run arbitrary shell commands within GHCi.) If `markets.json` isn't there, either move it there, or use `:cd` to move GHCi.⁴

⁴ `:cd` is a GHCi command. The missing `!` is intentional!

String theory

Haskell's built-in `String` type is a little silly. Sure, it's programmatically convenient to think of `Strings` as lists of characters, but that's a terrible, terrible way to store chunks of text in the memory of a computer. Depending on an application's need, there are several other representations of chunks of text available. This assignment will need two other representations: `ByteString` and `Text`.

The `ByteString` library helpfully (?) uses many of the same names for functions as the `Prelude` and `Data.List`. If you just import `Data.ByteString`, you'll get a ton of name clashes in your code. Instead, we use `import qualified ... as B`, which means that every use of a `ByteString` function (including operators) or type must be preceded by `B.`. Thus, to get the length of a `ByteString`, you use `B.length`. Even to mention the type `ByteString`, you must use `B.ByteString`.

`ByteStrings` come in several flavors, depending on whether they are lazy or strict and what encoding they use internally. Laziness is a story for another day, and we *really* don't want to worry about encodings. For now, use `Data.ByteString.Lazy.Char8` and everything will work out nicely.

`Text` is quite like `ByteString`: it *also* reuses a lot of familiar names. It *also* comes in two laziness flavors. We'll be using the strict flavor, which is provided in `Data.Text`. You also may want some I/O operations, so the import statements above include the `Data.Text.IO` module.

When working with non-`String` strings, it is still very handy to use the `"..."` syntax for writing `Text` or `ByteString` values. So, GHC provides the `OverloadedStrings` extension. This works quite similarly to overloaded numbers, in that every use of `"blah"` be-

comes a call to `fromString "blah"`, where `fromString` is a method in the `IsString` type class. Values of any type that has an instance of `IsString` can then be created with the `"..."` syntax. Of course, `ByteString` and `Text` are both in the `IsString` class, as is `String`.

A consequence of `OverloadedStrings` is that sometimes GHC doesn't know what string-like type you want, so you may need to provide a type signature. You generally won't need to worry about `OverloadedStrings` as you write your code for this assignment, but this explanation is meant to help if you get strange error messages.

Off to the market



<http://www.bayoucityoutdoors.com/ClubPortal/ClubStatic.cfm?clubID=36&pubmenuoptID=11318>

The `markets.json` file you have downloaded contains information about many (all?) of the Farmers' Markets that regularly take place throughout the USA, originally retrieved via <http://catalog.data.gov/dataset/farmers-markets-geographic-data>. That website produces the data in an Excel spreadsheet. I converted the spreadsheet to a comma-separated-values format (CSV) and then used <http://www.convertcsv.com/csv-to-json.htm> to get it into a JSON format. I chose JSON because the `aeson` JSON parser is more advanced yet easier to use than the CSV parser package, `cassava`.⁵

Take a look at the data by opening up the file in a text editor. You'll see that each market entry has many different fields, all with distinct names but of different types.

First, notice that the many Boolean values in the file are labeled with `"Y"` or `"N"`. As smart as `aeson` is, it doesn't know that `"Y"` cor-

⁵ I am giving you these details in case you want to look at other datasets.

responds to `True` and `"N"` corresponds to `False`. Your first task is to construct a `Value`⁶ that has been adjusted to have proper Booleans instead of `"Y"` and `"N"`.

One `aeson` function that parses JSON is called `eitherDecode`:

```
eitherDecode :: FromJSON a => ByteString -> Either String a
```

The `FromJSON` type class is also exported by the `aeson` package⁷. Its one method `parseJSON :: Value -> Parser a` (which you will *not* have to write in this assignment) says how to parse from JSON to the class type `a`. Thus, anything in the `FromJSON` type class can be parsed from a JSON file. Of course, parsing can fail, so `eitherDecode` returns either the desired value or a `String` containing error information.

A useful member of the `FromJSON` type class is `Value`. `Value` represents JSON syntax in a Haskell type. Check out its documentation.⁸ A JSON `Value` can be one of six things: an object (something in braces; a mapping from key names to other values), an array (something in brackets; a listing of JSON values), some text, a number, a Boolean value, or the special constant `null`. Look a little further down in the documentation to see the definitions for the types `Object` and `Array`.

An `Object` is a `HashMap Text Value` — that is, a way to get `Values` indexed by some `Text`. However, the details of `HashMap` aren't important at all for you. What *is* critically important is that there is an instance `Functor (HashMap k)`. That means that a valid type for `fmap` is `(a -> b) -> HashMap k a -> HashMap k b`.

An `Array` is a `Vector Value`. `Vector` is a type quite like normal lists but uses a different internal representation.⁹ Some operations on `Vectors` are faster than for lists; some are slower. However, the details of `Vector` aren't important at all for you. What *is* critically important is that there is an instance `Functor Vector`. That means that a valid type for `fmap` is `(a -> b) -> Vector a -> Vector b`.

Exercise 1 Write a function

```
ynToBool :: Value -> Value
```

that changes all occurrences of `String "Y"` to be `Bool True` and all occurrences of `String "N"` to be `Bool False`. No other part of the input `Value` should change.

Exercise 2 Write a function

```
parseData :: B.ByteString -> Either String Value
```

that takes in a `ByteString` containing JSON data and outputs either an error message or a `Value` that has been processed by `ynToBool`.

⁶ `Value` is a type from the `aeson` library. Hoogle does not search the `aeson` package by default, so you will have to access the package documentation on Hackage. Try this URL: <http://hackage.haskell.org/package/aeson-0.8.0.0>. There is a newer version uploaded (0.8.0.1), but for some reason, the documentation isn't there.

⁷ In case you haven't noticed, I'm using "package" and "library" interchangeably.

⁸ Ignore the `!s` in the documentation. They are strictness annotations, which are a story for another day. They don't affect you at all here.

⁹ Haskell lists are linked lists; `Vectors` are arrays.

Hint: This can be very short, if you use Either's Functor instance!

You can easily test this by saying, for example, `filedata <- B.readFile "markets.json"` in GHCi and then calling `parseData` on `filedata`.

The Market type

If you define a datatype `Market` appropriately, include deriving `Generic`, and say `instance FromJSON Market` with no `where` clause, you get an automatic parser for your datatype. For example, if you have

```
data Person = Person { name :: String
                      , age  :: Int }
    deriving (Show, Generic)
instance FromJSON Person
```

```
p :: Either String Person
p = eitherDecode "{ \"name\" : \"Richard\", \"age\" : 32 }"
```

You get that `p == Right (Person "Richard" 32)`.¹⁰ The `aeson` library uses the field names in the `Person` record (see the lecture notes about record notation) to figure out what the JSON tags should be. The order doesn't matter – `aeson` really is using the names.

¹⁰ The extra backslashes in the string above are because the string must contain quotes, and Haskell's way of putting quotes in strings is to use backslashes like you see here.

Exercise 3 Write a `Market` type, including the fields that interest you. At a minimum, include `marketname`, `x` (the longitude of the market), `y` (the latitude of the market), and `state`. Use `T.Text` to represent text. (`String` also works, but is less efficient.)

Then, write a function

```
parseMarkets :: B.ByteString -> Either String [Market]
```

that uses `parseData` and `fromJSON` (from the `aeson` package) to parse in the list of markets in the file.

Exercise 4 Write an I/O action

```
loadData :: IO [Market]
```

that loads the market data. In the event of a parsing failure, report the error using `fail :: String -> IO a`. (`fail` aborts an action, reporting an error to the user. It never returns, so it can be used no matter what `IO` type is expected. That's why it returns type `IO a`, for any `a`.)

Once this is defined, you can get your market data by saying `mkts <- loadData` in GHCi.

Interlude: an ordered-list monoid

Before we get to actually searching the loaded market data, it will be helpful to define a monoid for an ordered list. An ordered list, which we'll call `OrdList`, is a wrapper around lists (in the tradition of `First`, `Last`, `Sum`, and `Product`, all from the `Data.Monoid` module) that defines a `Monoid` instance which keeps the list ordered, from least to greatest. For example:

```
data OrdList a = OrdList { getOrdList :: [a] }
  deriving (Eq, Show)
  -- include this datatype in your code!

instance Ord a => Monoid (OrdList a) where ...
  -- you'll need to fill in the ellipses

evens :: OrdList Integer
evens = OrdList [2,4,6]

odds :: OrdList Integer
odds = OrdList [1,3,5]

combined :: OrdList Integer
combined = evens <> odds
```

Now, `combined` should have the value `OrdList [1,2,3,4,5,6]`, because the `(<>)` operator maintains the ordering invariant.

Exercise 5 Write the `OrdList` datatype and its `Monoid` instance.

Searching with Monoids

Now that you have a way of loading the market data, you need a way of searching through that data. Furthermore, it would be nice if the search mechanism is flexible enough to produce data in a `Monoid` of the caller's choice. Although there are, I'm sure, other queries you might want to do, all of our queries are going to center on searching for a market's name (the `marketname` field).

Throughout this section, we will be searching among the markets returning a variety of types. To avoid code repetition, it is helpful to use a type synonym. Include the following in your code:

```
type Searcher m = T.Text -> [Market] -> m
```

Thus, a `Searcher m` is a function that, when given the `T.Text` to search for in a `[Market]`, will produce an `m`.

Exercise 6 Write a function

```
search :: Monoid m => (Market -> m) -> Searcher m
```

that searches through the provided list of Markets for market names containing the given `T.Text` (`Data.Text.isInfixOf` will be useful here). With each found market record, use the function provided to convert the market record into the monoid of the caller's choice, and then combine all the individual results using `mappend` and `mconcat`.

Note that we can always expand type synonyms in Haskell. So, the type of `search` is fully equivalent to `Monoid m => (Market -> m) -> T.Text -> [Market] -> m`. This means that the definition for `search` may include up to three arguments, even though the type looks like it should take only one.

Hint: This function should not be very long. If it's getting long, you're probably doing something the wrong way. You may also want to check out the `intInts` example from the lecture notes.

Hint: Using an *as-pattern* may be helpful. Here is an example:

```
marketWithName :: Market -> (T.Text, Market)
marketWithName mkt@(Market { marketname = name }) = (name, mkt)
```

Note that `mkt` is matched against the whole market record, while the pattern-match also binds `name` to the market's name. The name is just one field in the record matched by `mkt`.

Exercise 7 Write a function

```
firstFound :: Searcher (Maybe Market)
```

that returns the first market found by a search, if any are found at all.

Like in the case for `search`, above, your `firstFound` function can be given arguments, even though the type looks like there should be no arguments. Unlike `search`, though, this one is definable without taking any arguments, with the right call to `search`.

Hint: The following function may be useful for all the searching exercises. Look at the type to figure out what it does:

```
compose2 :: (c -> d) -> (a -> b -> c) -> a -> b -> d
compose2 = (.) . (.)
```

Exercise 8 Write a function

```
lastFound :: Searcher (Maybe Market)
```

that returns the last market found by a search, if any are found at all.

Exercise 9 Write a function

```
allFound :: Searcher [Market]
```

that returns all the markets found by a search.

Exercise 10 Write a function

```
numberFound :: Searcher Int
```

that returns the number of markets found by a search.

Exercise 11 Write a function

```
orderedNtoS :: Searcher [Market]
```

that returns all the markets found by a search, ordered from northernmost to southernmost. You will need a wrapper around `Market` to choose an appropriate `Ord` instance. This exercise doesn't take too much code, but getting the structure right is intended to be a challenge.

You may find that your function takes a little while to run. As an optional extra, make it work more efficiently by adding a definition for `mconcat` to the `Monoid` instance for `OrdList` and make sure your search function uses `mconcat`. The default definition for `mconcat` puts elements together one by one, but you can write a custom one that maintains the ordering in a more efficient fashion.

Exercise 12 (*Optional*) Now that you've built the infrastructure to do queries on this dataset, see if you can find an interesting detail in the data. It should be fairly easy at this point to make a variety of queries. Tell us something we didn't know about farmer's markets!