Practical Data Processing With Haskell

Ozgun Ataman

SOOSTONE

November 14, 2012

A bit about the speaker

- Electrical Engineering, Biomedical Engineering, Business School
- Later: "Management Consulting" with a strong flavor of analytics
- Coding for 15 years, Haskell for 4 years
- Used to use Ruby and Python for everything
- First started with Haskell on data/analytics/simulation problems
- Founded Soostone; using Haskell for almost everything
- Core contributor to Snap Framework
- Author of a number of libs on Hackage (and some yet to be released)

Motivation for this talk: Data processing is a great practical place to start using (and learning) Haskell

- We will assume very little familiarity with Haskell
- Can't teach the whole language; there are very good other resources
- Will try to point at some simple but practically useful real-world scenarios
- Will expose some Haskell syntax along the way
- Hopefully you'll feel like giving Haskell a shot next time you run into a similar challenge

Why Haskell? (Every talk must have one)

- Type system doubles as a design language, crystallizes thoughts
- Catch errors early, refactor aggressively (vs. Ruby/Python)
- Purity is a huge win for long-lived, "can't fail" code
- Stay at a very high level, yet still get solid performance
- Testing is even better, QuickCheck et al. are mesmerizing
- Ridiculously simple multi-core concurrency
- Promising future for parallel algorithms

"Comma Delimited Values" is a ubiquitous format that necessitates some frequent, boring tasks for the analyst

Common Data Processing Tasks

- Simple Tasks
  - Tidy up a messy data feed before stuffing into a SQL database
  - Transform a stream of tabular data (CSV) as a "pre-processing" step
  - Connect to a JSON/XML API and convert to clean CSV for multi-purpose use later

"Comma Delimited Values" is a ubiquitous format that necessitates some frequent, boring tasks for the analyst

Common Data Processing Tasks

- Simple Tasks
  - Tidy up a messy data feed before stuffing into a SQL database
  - Transform a stream of tabular data (CSV) as a "pre-processing" step
  - Connect to a JSON/XML API and convert to clean CSV for multi-purpose use later
- "Sky Is The Limit"
  - Load strongly typed data for use in algos, simulations, etc.
  - Develop re-usable data processing and task automation tools

We will try to hit one example in each category.

Getting started with Haskell

Installing Haskell

1. Download Haskell Platform at http://www.haskell.org/platform/
2. `cabal update`
3. `cabal install [insert package name here]`

Getting started with Haskell

Installing Haskell

1. Download Haskell Platform at http://www.haskell.org/platform/
2. `cabal update`
3. `cabal install [insert package name here]`

Learning Resources

1. `http://learnyouahaskell.com/`
2. `http://book.realworldhaskell.org/`
3. `http://en.wikibooks.org/wiki/Haskell`
4. `#haskell` on freenode

A few summarizing words about Haskell

- Haskell is purely functional - separates side effects from equational logic
- Haskell is non-strict - you don't control order of evaluation
- Haskell is staticly typed
- Haskell is strongly typed - conversions are explicit
- Haskell compiles to native code
- GHC is pretty much the de-facto compiler for (public) real-world work
- The RTS can map its lightweight threads onto several OS threads - no global lock like Python or Ruby
- Several language extensions are commonly used to increase expressiveness

A real simple example: Let's parse some CSV

```haskell
module Main where

import qualified Data.ByteString.Char8 as B
import Data.ByteString.Char8 (ByteString)

-- type synonyms are helpful for clarity
type Field = ByteString
type Row = [Field]
type CSV = [Row]

-- parsing of CSV is a pure conversion
-- from string to our defined CSV type
parseCSV :: ByteString -> CSV
parseCSV string = (map (B.split ',') . B.lines) string

-- all I/O occurs separately, note the IO in type
main :: IO ()
main = do
    contents <- B.readFile ``Test.csv''
    print (parseCSV contents)
```

A real simple example: Let's parse some CSV

```
module Main where

import qualified Data.ByteString.Char8 as B
import Data.ByteString.Char8 (ByteString)

-- type synonyms are helpful for clarity
type Field = ByteString
type Row = [Field]
type CSV = [Row]

-- parsing of CSV is a pure conversion
-- from string to our defined CSV type
parseCSV :: ByteString -> CSV
parseCSV string = (map (B.split ',') . B.lines) string

-- all I/O occurs separately, note the IO in type
main :: IO ()
main = do
    contents <- B.readFile ''Test.csv''
    print (parseCSV contents)
```

What about different delimeters, line endings, text quotation?

Don't reinvent the wheel: Use a CSV library

There are several good ones around these days (wasn't always so):

- bytestring-csv: Simple, similar to what we have here
- csv-conduit: Fast, flexible, stream processing CSV lib (by yours truly)
- cassava: Fast, easy to use recent release by Johan Tibell

Don't reinvent the wheel: Use a CSV library

There are several good ones around these days (wasn't always so):

- bytestring-csv: Simple, similar to what we have here
- csv-conduit: Fast, flexible, stream processing CSV lib (by yours truly)
- cassava: Fast, easy to use recent release by Johan Tibell

Example: Just read a file

```
import Data.CSV.Conduit

-- You can use ''Text'' for proper uni
code support
type MapRow Text = Map Text Text

readCSVFile
  :: CSVSettings -- Specify delimiter and text quotation
  -> FilePath -- Point at a file
  -> IO [MapRow Text]
```

Flexibility is important in real-world usage; you'll run out of options fast if tied to the official RFC

We need something that can alter CSV format on both sides of I/O:

```
-- All CSV.Conduit operations take these options
data CSVSettings = CSVS {
    csvSep :: Char
    -- ^ Field delimeter
  , csvQuoteChar :: Maybe Char
    -- ^ Text wrapper
  , csvOutputQuoteChar :: Maybe Char
    -- ^ Output text wrapper
  , csvOutputColSep :: Char
    -- ^ Output delimeter
  } deriving (Read, Show, Eq)


-- We can start with defaults and just tweak the part we need.
let mySettings = defCSVSettings { csvSep = '~' }
```

Let's sustainably solve a common problem by creating a command-line utility

Problem Statement

Data from many legacy sources often come with bizarre delimeters, no proper text quotation and with extraneous white space.

Let's sustainably solve a common problem by creating a command-line utility

Problem Statement

Data from many legacy sources often come with bizarre delimeters, no proper text quotation and with extraneous white space.

```
MAZDA6          ~23500.00    ~00123    ~
SUBARU IMPREZA  ~33420.00    ~00078    ~
```

Let's sustainably solve a common problem by creating a command-line utility

Problem Statement

Data from many legacy sources often come with bizarre delimeters, no proper text quotation and with extraneous white space.

```
MAZDA6          ~23500.00    ~00123    ~
SUBARU IMPREZA  ~33420.00    ~00078    ~
```

It's hard to believe, but many "data analysts" spend hours(!!) cleaning up datasets using all string-typed SQL tables and ad-hoc queries.

Let's sustainably solve a common problem by creating a command-line utility

Problem Statement

Data from many legacy sources often come with bizarre delimeters, no proper text quotation and with extraneous white space.

```
MAZDA6          ~23500.00    ~00123    ~
SUBARU IMPREZA  ~33420.00    ~00078    ~
```

It's hard to believe, but many "data analysts" spend hours(!!) cleaning up datasets using all string-typed SQL tables and ad-hoc queries.

Mission

Create a command line tool that can do this "automatically" for us:

- Be flexible in field separator and text quotation character
- Be able to operate on really large files
- Strip each field of any surrounding white-space

In Haskell, we often start with the types

We need something like:

```
-- Take a file, clean it up, output into another file
procFile :: CSVSettings -> FilePath -> FilePath -> IO ()

-- We may choose to drop rows or emit multiple rows per row during th
-- transformation
fixRow :: Row -> [Row]

-- Do all the needed fixes on each column here
fixField :: Text -> Text
```

In Haskell, we often start with the types

We need something like:

```
-- Take a file, clean it up, output into another file
procFile :: CSVSettings -> FilePath -> FilePath -> IO ()

-- We may choose to drop rows or emit multiple rows per row during th
-- transformation
fixRow :: Row -> [Row]

-- Do all the needed fixes on each column here
fixField :: Text -> Text
```

A simple implementation:

```
fixfield = T.strip      -- drop whitespace

-- drop empty rows, fix each column otherwise
fixRow [x] = case fixField x of "" -> []
                                x' -> [x']
fixRow xs = [map fixField xs]
```

What else can we do?

Often there will be specific columns that require special treatment. You may need to parse a month name or split each "search term" out into its own row.

Split each search term into its own row:

```
-- take each row, split the terms
-- and add as an additional column into that row
procRow :: MapRow Text -> [MapRow Text]
procRow m = map ins pieces
  where
    ins = M.insert "term" v      -- insert new term into dictionary
    terms = m ! "terms"          -- lookup a field from dictionary
    pieces = T.split ' ' terms   -- tokenize terms using whitespace
```

Let's package it all up

```haskell
module Main where
import System.Environment
import Data.CSV.Conduit

-- Map our fixRow function over the rows of the given CSV file
procFile set input output = mapCSVFile set fixRow input output

-- Read arguments from the command line and call procFile
main = do
  args <- getArgs
  case args of
    (fi : fo : sep : quote : _) -> do
        let set = defCSVSettings { csvSep = head sep
                                 , csvQuoteChar = Just (head quote) }
        procFile set fi fo
        print ''Processing complete!''
    _ -> error ''You must provide exactly 4 arguments!''
```

We now have a fast, reusable executable flexible enough to become the first step in any data analysis exercise.

Going further: Automate new SQL table creation and ongoing import of incoming data

Problem Statement

Creating new SQL tables for ad-hoc analysis of a 235-column dataset is a HUGE pain, especially if you need to do it 3 times a day.

Going further: Automate new SQL table creation and ongoing import of incoming data

Problem Statement

Creating new SQL tables for ad-hoc analysis of a 235-column dataset is a HUGE pain, especially if you need to do it 3 times a day.

Mission

What if we could automatically deduce column data types, size them right and generate SQL for the table creation?

We will not go into details, but want to highlight some parts that demonstrate why Haskell shines

Algebraic Data Types are a big help in modeling the problem domain

```
data Field
    = FInt !Integer !Integer
    | FDouble !Double !Double
    | FVarStr !MaxLen
    | FText !MaxLen
    | FDate
    | FDateTime
    | FBool
    | FBlank
    deriving (Show, Eq, Ord, Read)

-- try parsing each type in an order that makes sense
identifyField :: String -> Field
```

Algebraic Data Types are a big help in modeling the problem domain

```
data Field
     = FInt !Integer !Integer
     | FDouble !Double !Double
     | FVarStr !MaxLen
     | FText !MaxLen
     | FDate
     | FDateTime
     | FBool
     | FBlank
     deriving (Show, Eq, Ord, Read)

-- try parsing each type in an order that makes sense
identifyField :: String -> Field

-- As we stream over sample rows, we will maintain best-guess types
type IDMap = HM.Map String Field

-- New evidence may change our guess.
(<>) :: Field -> Field -> Field
FInt mn1 mx1 <> FInt mn2 mx2 = FInt (min mn1 mn2) (max mx1 mx2)
_            <> FText l      = FText l
```

Result: A command-line utility we're calling 'sqlimport'

'sqlimport' is a full-fledged command line program:

```
➜ ~ sqlimport
Usage: sqlimport COMMAND

Available options:
  -h,--help                Show this help text

Available commands:
  gentable                 Sample rows from file and produce SQL that would create a database table.
  import                   Import all columns that match target table.
```

Result: A command-line utility we're calling 'sqlimport'

'sqlimport' is a full-fledged command line program:

```
➜ ~ sqlimport
Usage: sqlimport COMMAND

Available options:
  -h,--help                Show this help text

Available commands:
  gentable                 Sample rows from file and produce SQL that would create a database table.
  import                   Import all columns that match target table.
```

It can define table schema (or import directly) for MySQL and Postgres:

```
➜ ~ sqlimport gentable
sqlimport - tablegen - convenient SQL table creation from CSV files

Usage: sqlimport gentable (-i|--input INPUT_FILE) (-o|--output OUTPUT_FILE) [-k|--size ROWS] [-t|--type SQL_TYPE] (-l|--table DB_TABLE)
  Sample rows from file and produce SQL that would create a database table.

Available options:
  -h,--help                Show this help text
  -i,--input INPUT_FILE    Input file to read from
  -o,--output OUTPUT_FILE  Output file to write SQL text
  -k,--size ROWS           Sample K rows from the input file
  -t,--type SQL_TYPE       Either 'Postgres' or 'MySQL'
  -l,--table DB_TABLE      Name of the SQL table to be created
➜ ~
```

Thank you for listening!

Any Questions?