God Damn Monads: Dropping the Mic on Wrangell

Dana Iltis, Kenan Nalbant, and Donald Pinckney

University of California, Davis
{driltis, kanalbant, djpinckney}@ucdavis.edu

ABSTRACT

Managing data from various document types (e.g. txt, csv files) often requires manual work by a user. We present Wrangell, a domain-specific language for common data manipulation tasks.

Run from the command line, Wrangell is specifically designed to modify tables of data wherein columns are separated by a user-specified delimiter. Wrangell has functions to read in a file, parse the data, run a transformation on the table, and write the modified table to an output file.

We implemented Wrangell in Haskell, a functional programming language. We chose Haskell for its powerful type system which allows a large amount of control over how operations behave when handed various data types. Furthermore, as a functional programming language, Haskell is a good choice for tasks which depend on running transformations on data.

We based Wrangell on Scheme, a Lisp-like language known for its simplicity. Wrangell's simplicity is desirable as it would facilitate ease of integration into potential future data management projects.

1. INTRODUCTION

For our project, we wanted to design and implement a small language. Haskell was an attractive tool because of its type system in which data types are immutable and are defined algebraically. Furthermore, our team members had either a beginner or intermediate level of Haskell knowledge, so we saw this as an opportunity to learn something new.

To keep the scope of our language's features manageable, we chose to gear our it to a specific domain. Because Haskell is a functional language, we thought that *data-wrangling*, with its emphasis on data transformations, would be a good application for our language implemented in Haskell. We began creating Wrangell, a DSL for "WRANGling" data via "haskELL".

We focused our desired data-wrangling tasks on one format: a table of data in a csv file, wherein we recognize rows via lines of the file, and columns per a user-specified delimiter. The following operations would initially be included in Wrangell:

1. Read in an input filename, output filename, () , ()
2. Read in and parse a csv file per a filename
3. Run some transformation on the data in the file
4. Write the transformed data to an output file

A motivating example for Wrangell could be the following: A small business has many csv files (say, one for each year the company has been active) with customer data. Each file has a column for various items including the customer's name, address, phone number, and credit card number. Now say the business wanted to contract some advertising company to mail a promotional catalog to each of its customers. In this case, the business would want to share its customer database with the advertising company. However, sharing customer credit card information with the advertising company would be both unnecessary and dangerous. In this case, the business would want an easy way to make copies of the customer information files with the credit card information column removed.

2. APPROACH

Because our team memebers only had beginner or intermediate levels of Haskell knowledge, we chose to keep Wrangell's data management operations fairly simple. As a result, Wranagell's application subjects are limited to tables of data wherein columns were separated by either a comma or a user-specified delimiter.

We determined big-step semantics for Wrangell's operations (Figure 1). Then we planned to apply Wrangell's data manipulation features to a variety of csv files with a data table formatted per the example in Figure 2.

We decided that Wrangell should support the following transformation operations:

1. Remove a column
2. Insert a column
3. Remove a row
4. Insert a row

Blah blah blah. other stuff about the approach.

1. Table

$$\Sigma = \bigcup_{n=1,r=1}^{\infty} \{M | t_i \epsilon T, m_{*i} \epsilon V(t_i : r), l_i \epsilon \{Strings\}, \forall l_i \nexists l_j s.t.\ l_i = l_j \wedge j \neq i\}$$

$\sigma[M] = (t_1, t_2, ..., t_n), (m_{*1}, m_{*2}, ..., m_{*n}), (l_1, l_2, ..., l_n)$

2. Drop Column

$$\frac{< name, \sigma > \Downarrow l_i}{< dropCol\ name, \sigma > \Downarrow \sigma'}$$

$\sigma'[M] = \{(t_1, t_2, ..., t_{i-1}, t_{i+1}, ..., t_n), (m_{*1}, m_{*2}, ..., m_{*i-1}, m_{*i+1}, ..., m_{*n}), (l_1, l_2, ..., l_{i-1}, l_{i+1}, ..., l_n)\}$

3. Drop Row

$$\frac{m'_{*i} = \{m_{ji} | < \lambda m_{ji}, \sigma > \Downarrow False\}}{< dropRow\ \lambda, \sigma > \Downarrow \sigma'}$$

$\sigma'[M] = \{(t_1, t_2, ..., t_n), (m'_{*1}, m'_{*2}, ..., m'_{*n}), (l_1, l_2, ..., l_n)\}$

4. Insert Column

$$\frac{j \epsilon [1, n+1],\ name\ \epsilon \{l_1, l_2, ..., l_n\}}{< insertCol\ type\ name\ j, \sigma > \Downarrow \sigma'}$$

$\sigma'[M] = \{(t_1, t_2, ..., t_{j-1}, type, ..., t_n), (m_{*1}, m_{*2}, ...0()..., m_{*n}), (l_1, l_2, ..., name, ..., l_n)\}$

Figure 1: Big-step semantics for Wrangell's table manipulation functions

Filename: twinPeaksPeople.csv

Name, Age,Gender, Favorite Food, School, Social Security Number
Dale, 40, M, Coffee, Dartmouth 111111111
Audrey, 18, F, Coffee, Twin Peaks High, 888888888
Laura, 17, F, Cereal, Twin Peaks High, 000000000
Bobby, 17, M, Bacon, Twin Peaks High, 333333333
Hawk, 34, M, Doughnuts, Brandeis, 111223333
Ben, 48, M, Brie sandwich, USC, 444556666
Hank, 34, M, Dominoes, Prison GED Program, 888116666
Leo, 32, M, Raw Hamburger, Sarah Lawrence, 000996666
Jacoby, 51, M, Coconuts, USC, 000997777
Blackie, 43, F, Shirley Temple, School of Lyfe, 999771111

Figure 2: Example of a Wrangell-compatible data table

```
-- datatypes.hs --
import Data.IORef
.
.
.
data Table' =
Table' { rows :: [[WVal]],
         format :: [WType],
         labels :: [String],
         delimiter :: String}
type Table = IORef Table'
-- TODO: This will be a bit different
.
.
.
--creates a new empty table context
emptyTable :: IO Table
emptyTable = new IORef Table' {
  rows   = [[]],
  format = [],
  labels = [],
  delimiter = ","
}
```

Figure 3: Creation of the 'Table' data type within the IO Monad

```
-- functions.hs --
.
.
.
funcTable :: [(FuncDef, [WVal] -> WVal)]
functable = [
   (("if", [TBool, TFloat, TFloat]), if'),
   (("if", [TBool, TBool, TBool]), if'),
   (("car", [TList [TIntegral,TIntegral]]),
      car' )]
.
.
.
if' :: [WVal] -> WVal
if' [condition, t, f] =
    if unpackBool condition then t else f
```

Figure 4: Inflexible type system: Type-checking occurs in funcTable, no type-checking in ''if' function definition

3. IMPLEMENTATION AND RESULTS

We ran into several challenges when writing our Wrangell code in Haskell. Most of our challenges centered on our need to mimic imperative behavior in Haskell, a functional language.

3.1 Modifying Data in an Immutable State

Our first challenge was getting Wrangell to store information about the contents of the input file. This conflicts with the idea that a functional language limits its environment to an "immutable state". For example, Wrangell would need to store the parsed data table from each input file. Additionally, Wrangell needs to modify the data table. Our solution was to perform operations within the *IO Monad*, an abstract data type which provides a system wherein we can build "composite actions". This allows us to mimic imperative behavior without requiring mutability of variables. The below example demonstrates our use of the IO monad during construction of an empty data table for later use: Because emptyTable is in the IO monad, we can later () as follows:

```
--blah blah blah
```

Without defining emptyTable as an IORef, we would have (encountered an error?) above.

Additionally, Use of the IO Monad via IORef also allows Wrangell to be used to make âĂIJhelper functionsâĂİ, âĂęâĂę

We describe monads and IO further in âĂŸRelated WorkâĂŹ below.

3.2 Type System Flexibility

The second challenge was creating a type system which was flexible enough to achieve our desired level of expressiveness while still behaving within the confines of Haskell's type system. We initially defined functions in 'functable' as follows: In this strategy we use funcTable to constrain exactly what inputs a function can accept. However, for the car function (which computes the first item of a list), in order to describe lists of varying length and types, we would have to include a line (an instance of the funcTable? class?) for every possible combination of lengths and types!

Our solution (Figure 5) was to move the type-checking of inputs into the functions themselves. This allows more flexibility in funcTable. However, because it would now be possible to pass incompatible data to a function, we needed a way for functions to handle potential errors. Our solution, described in related work, was to implement type-safe exception handling via the Either monad.

3.3 Getting Wrangell to Read in csv Files

We used existing open source code from data.csv [1] to read in and parse the csv data files.

4. RELATED WORK

Monads
Monad transformers
The IO monad (IORef)
Application to other file formats (hdf5)
data.csv

5. CONCLUSIONS

Wrangell

Hypothetical future improvements of Wrangell might include improvement in label analysis, a feature which would allow users to identify columns purely via their label. For example, an operation meant to aggregate email addresses for all customers since 1995 would need to handle customer data files wherein the column "email address" is not present (e.g. files prior to 2002). Additionally, it would also be useful if Wrangell could handle files with inconsistent ordering of columns. Finally, we would want Wangell to be able to handle data tables with an unknown number of columns.

6. REFERENCES

```
-- functions.hs --
funcTable' ::
   [(String, [WVal] -> ThrowsError WVal)]
FuncDef
functable = [
    ("if", if')
    ("car", car)]
.
.
.
if' :: [WVal] -> ThrowsError WVal
if' ifComps = do
  --error checking
  checkLength (==) 3 ifComps
  checkType TBool $ head ifComps

  --checks that the types
  --  of the consequents match
  if getType t == getType f
  then return $ List $ tail ifComps
  else throwError $ TypeError
       "expected types to match,
        found" $ List $ tail ifComps

  --does the actual if computation
  if unpackBool cond
     then return t
     else return f

  where cond = head ifComps
        t    = ifComps !! 1
        f    = ifComps !! 2
.
.
.
car :: [WVal] -> ThrowsError WVal
car [List (x : _)] = return x
car [badArg]       =
     throwError $ TypeError "pair" badArg
car badArgList     =
     throwError $ NumArgs 1 badArgList
```

Figure 5: type-checking now takes place in if' and car function definitions. No restrictions on types in funcTabe' definitions.

[1] jgoerzen/missingh. URL `https://github.com/jgoerzen/missingh`.