

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 members only had beginner or intermediate levels of Haskell knowledge, we chose to keep Wrangell's data management operations fairly simple. As a result, Wrangell'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

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

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

2. Drop Column

$$\frac{\langle name, \sigma \rangle \Downarrow l_i}{\langle dropCol\ name, \sigma \rangle \Downarrow \sigma'}$$

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

3. Drop Row

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

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

4. Insert Column

$$\frac{j \in [1, n+1],\ name \in \{l_1, l_2, \dots, l_n\}}{\langle insertCol\ type\ name\ j, \sigma \rangle \Downarrow \sigma'}$$

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

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

LATER USE OF THE TABLE CONTEXT/INSTANCE?

Figure 4: placeholder

3. IMPLEMENTATION AND RESULTS

WE based Wrangell on Scheme..... In coding Wrangell, most of our challenges centered on our need to mimic imperative behavior in Haskell, a functional language. We needed to store and modify table data, a goal which seems to contradict Haskell’s ‘immutable data’ requirement. Additionally, we needed to implement a type system which was flexible enough to represent lists for our table columns/rows. Finally, we needed to give Wrangell the ability to read and write csv files.

3.1 Modeling Wrangell After Scheme

We based Wrangell on Scheme, a Lisp dialect.blah blah blah. For more on Scheme see ‘Related Work’ below.

3.2 Working with Haskell’s Immutable Data

In Haskell, data is immutable (see ‘Immutable Data and Monads’ in ‘Related Work’). Yet, we needed Wrangell to store and modify information about our original and transformed data tables. Our solution was to define tables using the *data.IORef* package. With this package we constructed our *Table* data type to be an *IORef* of(WAIT WHAT ”TODO this will be a bit different”). The code in Figure 3 demonstrates our use of the IO monad during construction of an empty data table for later use (Later use example in Figure 4) . Without defining emptyTable as an IORef, we would have (encountered an error?) during later use.

Because emptyTable is in the IO monad, we can later () as follows:

```

— 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 5: Inflexible type system: Type-checking occurs in funcTable, no type-checking in ‘if’ function definition

3.3 Type System Flexibility

Wrangell needed a type system which was flexible enough to achieve our desired level of expressiveness, particularly when describing lists necessary to represent data table columns/rows. We initially defined type restrictions directly in ‘funcTable’ (Figure 5). 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 6) 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.

3.4 Reading and Writing csv Files

For Wrangell’s reading, parsing, and writing csv files, we used some existing code from the public GitHub repository MissingH [1]. Specifically we used CSV.hs, which uses Data.csv to

More description of how Wrangell reads in files....

4. RELATED WORK

4.1 Scheme

Scheme is a Lisp dialect. It was useful to our construction of Wrangell because...

4.2 Monads in Haskell

Monads were one of the more difficult concepts for us to implement. ...blah blah blah... We used the IO monad to facilitate modification of our data tables.

4.3 MissingH, Data.csv, CSV.hs

MissingH is a publicly available repository of Haskell utility functions. We used CSV.hs to read/write csv files.....

```

— 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 6: Flexible type system: Type-checking now takes place in `if'` and `car` function definitions. No restrictions on types in `funcTable'` definitions.

CSV.hs works like this: ...

The IO monad (IORef) *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. Application to other file formats (hdf5)

data.csv

5. CONCLUSIONS

Wrangell

Hypothetical future improvements to 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 Wrangell to be able to handle data tables with an unknown number of columns.

6. REFERENCES

- [1] J. Goerzen. `missingh/src/data/csv.hs`. <https://github.com/jgoerzen/missingh/blob/master/src/Data/CSV.hs>, 2016.