Wrangell: A Data-Wrangling DSL in Haskell

Dana Iltis, Kenan Nalbant, and Donald Pinckney

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

ABSTRACT

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

In its current iteration Wrangell is distributed as a command line tool and is specifically designed for common sorts of data-preprocessing tasks. Wrangell is designed from the ground up to be easily extensible with regard to the kinds of input and output files that may be desired. In addition the syntax Wrangell script is simple yet flexible enough to allow for new transformations on data to be defined as needed by end-users.

We implemented an interpreter for 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 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. Our resulting project, Wrangell, is 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 were to be supported by Wrangell:

1. Read in an input file and convert to a representation internal to Wrangell
2. Run some transformation on the internal representation of the data
3. Write the transformed data to an output file (which can be of a different file type)

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. Such a transformation should be supported by Wrangell.

2. APPROACH

2.1 Wrangell-Compatible Data

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

At present Wrangell supports simple file types for input and output such as csv files, or character delimited files. However the Wrangell interpreter is equipped such that if another user were to define a function to transform input file types to the intermediate representation, and a function to transform from the intermediate representation to the output file format, that file format would then be fully supported with no further modifications to the interpreter being necessary.

2.2 Semantics of Wrangell

We decided that Wrangell should support the following transformation operations (See big-step semantics in Figure 2):

1. Remove a column
2. Insert a column
3. Remove a row
4. Insert a row
5. Run transformations on individual columns of data

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 1: Example of a Wrangell-compatible data table

1. Table

$$\Sigma = \bigcup_{n=0,r=0}^{\infty} \{M | t_i \in T, m_{*i} \in V(t_i : r), l_i \in \{Strings\}, \forall l_i \nexists l_j s.t.\ l_i = l_j \wedge j \neq i\}$$

$\sigma[M] = (t_0, t_1, ..., t_n), (m_{*0}, m_{*1}, ..., m_{*n}), (l_0, l_1, ..., l_n)$

2. Drop Column

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

$$\frac{i \in [0, n]}{< dropCol\ i, \sigma > \Downarrow \sigma'}$$

$\sigma'[M] = \{(t_0, t_1, ..., t_{i-1}, t_{i+1}, ..., t_n), (m_{*0}, m_{*1}, ..., m_{*i-1}, m_{*i+1}, ..., m_{*n}), (l_0, l_1, ..., l_{i-1}, l_{i+1}, ..., l_n)\}$

3. Drop Row

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

$\sigma'[M] = \{(t_0, t_1, ..., t_n), (m'_{*0}, m'_{*1}, ..., m'_{*n}), (l_0, l_1, ..., l_n)\}$

4. Insert Column

$$\frac{j \in [0, n+1] \wedge\ label\ \notin \{l_1, l_2, ..., l_n\}}{< insertCol\ type\ label\ j, \sigma > \Downarrow \sigma'}$$

$\sigma'[M] = \{(t_0, t_1, ..., t_{i-1}, type, ..., t_n), (m_{*0}, m_{*1}, ..., m_{*i-1}, m_{*i}..., m_{*n}), (l_0, l_1, ..., l_{i-1}, label, ..., l_n)\}$

5. Transform Column

$$\frac{< label, \sigma > \Downarrow l_i\ \wedge\ m'_{*i} = \{m'_{ji} | < \lambda m_{ji}, \sigma > \Downarrow m'_{ji}\}}{< transformCol\ \lambda\ label, \sigma > \Downarrow \sigma'}$$

$$\frac{i \in [0, n]\ \wedge\ m'_{*i} = \{m'_{ji} | < \lambda m_{ji}, \sigma > \Downarrow m'_{ji}\}}{< transformCol\ \lambda\ i, \sigma > \Downarrow \sigma'}$$

$\sigma'[M] = \{(t_0, t_1, ..., t_n), (m_{*0}, m_{*1}, ..., m_{*i-1}, m'_{*i}, ..., m_{*n}), (l_0, l_1, ..., l_n)\}$

Figure 2: Big-step semantics for Wrangell's table manipulation functions

```
-- datatypes.hs --

-- Define the ThrowsError to exist
--    within the Either monad
type ThrowsError = Either WError
.
.
.

-- main.hs --

--Left signals error, Right a proper parse
readOrThrow ::
  Parser a -> String -> ThrowsError a
readOrThrow parser input =
  case parse parser "wrangell" input of
    Left err  -> throwError $ Parser err
    Right val -> return val
```

Figure 3: Wrangell's use of the Either monad to define behavior in successful and unsuccessful cases in the readOrThrow function which parses user input

```
-- datatypes.hs --

type IOThrowsError = ExceptT WError IO
.
.
.

-- DataOperations.hs --
checkAllUnique ::
  [String] -> [WVal] -> IOThrowsError WVal
checkAllUnique labels wlabels =
  if allUnique labels
    then return $ Unit
    else throwError
         $ FormatSpec
           "Labels should be all unique:
             " wlabels
```

Figure 4: IOThrowsError data type created via the ExceptT monad transformer. IOThrowsError later used in checkAllUnique which depends on functionality from both the Either and the IO monad.

2.3 Monads in Haskell

A significant hurdle we faced when implementing the interpreter for Wrangell was the fact that the user could define new constants or functions on the fly (i.e. mutating the execution environment of Wrangell) and also mutate the table. To carry out activities like storing and modifying table data, we would need to use various Haskell *monads*, a 'conceptual structure' [5] in Haskell which, for our purposes in Wrangell, would allow us to chain together multiple computations (e.g. a sequence of functions, or a passing of the result of one function to another).

In Wrangell, we planned to use the built-in *Either* monad to manage errors such as the passing of the wrong data type to a function. For items in the Either monad, custom behavior can be defined for 2 cases: Left, wherein an error occurred, and Right, wherein the result was successfully parsed. We planned to use this to attach error descriptions for various data types.

In Figure 3 , we define the data type 'ThrowsError' to be in the Either monad, meaning any variable of type 'ThrowsError' will also be within the Either monad. We then use the Either monad Right/Left functionality in the readOrThrow user input-parsing function to define the behaviors in the Left and Right cases (also in Figure 3).

2.4 The IO Monad, Monad Transformers, and exceptT

We use the IO ('input/output') monad to handle any sort of input/output activities that the Wrangell interpreter might need. The type-safe error handling is accomplished with the *ExceptT* monad transformer. Monad transformers [3] are a construct which allow us to easily combine our monadic IO code and error handling code into a new composite monad which can handle both IO and errors. The IOThrowsError data type, for example, was inspired by an example we found in a Haskell tutorial [1]. Construction of the IOThrowsError data type in Figure 4 uses the monad transformer *ExceptT*. ExceptT incorporates the functionality of the Either monad and facilitates application of error handling to Wrangell errors (WError) in the IO monad. For example, Figure 4 also illustrates sample usage of IOThrowsError in the checkAllUnique function. checkAllUnique, which checks if the column labels provided to Wrangell are unique, uses throwError which provides a type-safe way of dealing with exceptions.

3. IMPLEMENTATION AND RESULTS

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

Wrangell's syntax is heavily inspired by Scheme's (a Lisp dialect) syntax. For more on Scheme see 'Related Work' below.

3.2 Working with Haskell's Immutable Data

In Haskell, data is immutable. 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 the data type Table'. The code in Figure 5 demonstrates our use of the IO monad during construction of an empty data table for later use. Packaging data in an IORef allows for very easy imperative style data manipulation through two key functions, readIORef and writeIORef, which behave exactly in the manner that their name implies.

3.3 Type System Flexibility

Wrangell needed a type system which was flexible enough to achieve our desired level of expressiveness, particularly

```
-- datatypes.hs --
import Data.IORef
.
.
.
data Table' =
  Table' { rows :: [[WVal]],
           format :: [WType],
           labels :: [String],
           outFileType :: FileType,
           outFileName :: String}
type Table = IORef Table'
.
.
.
--creates a new empty table context
emptyTable :: IO Table
emptyTable = newIORef Table' {
  rows    = [[]],
  format = [],
  labels = [],
  outFileType = File,
  outFileName = "" }
```

Figure 5: Creation of the 'Table' data type within the IO Monad

when describing lists necessary to represent data table columns/rows. We initially defined type restrictions directly in 'funcTable' (Figure 6). 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 definitions for every possible combination of lengths and types! Our solution (Figure 7) 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 was to implement exception handling via the exceptT monad transformer mentioned earlier. An additional benefit of the approach we eventually went with is that when an operator or function is exposed by our function table, any expressions defined in terms of these functions inherit the polymorphism of Haskell's type system as a consequence of the way things are structured.

3.4 Reading and Writing csv Files

For Wrangell's reading, parsing, and writing of csv files, we used existing code from the public GitHub repository MissingH [2]. Specifically we used CSV.hs, which uses Data.csv to parse comma-separated strings into lists of strings.

To support potential extension of Wrangell we wrote code which converts input file formats to an intermediate Wrangell table. Additionally we wrote code to convert from the intermediate Wrangell table to an output file format. Because Wrangell's data transformations operate on the intermediate table, this allows Wrangell to be applicable to a variety of file formats. This means that, hypothetically, a future iteration of Wrangell could read in a csv file, and output, say, a txt file.

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

3.5 User Defined Functions

A user can supply their own functions using the built-ins that are exposed to the Wrangell run-time which allows for the user to define any sort of arbitrary function. Users can then apply these functions to do transformations on the input data.

4. RELATED WORK

4.1 Scheme

Scheme is a Lisp dialect [6]. It was a fairly natural choice for us to heavily model Wrangell's syntax on Scheme's due to both its simplicity and the ease with which one can create a parser for it.

4.2 Monads in Haskell

As mentioned above, Wrangell needed to emulate imperative behavior in order to track and modify data. Previous work on monads and monad transformers [3] was useful in constructing Wrangell's capabilities for data table information modification and error handling.

4.3 MissingH, Data.CSV, CSV.hs

MissingH [2] is a publicly available repository of Haskell utility functions. We modified Data.CSV into a file called CSV.hs which is better suited to our purposes (Data.CSV only allowed for commas as delimiters).

4.4 DSLs

As a Domain Specific Language, Wrangell was intended to be a fast, simple tool to carry out data management tasks. Paul Hudak's paper on Domain Specific Languages champions the usefulness of program written in DSL's because of these exact qualities [4]. An idea not yet addressed in our discussion of Wrangell is Hudak's point that DSL's are more usable by those without significant programming expertise. Wrangell could thus be a starting point for a data-wrangling product used in industries where users are less likely to have programming skills.

5. CONCLUSIONS

```
-- functions.hs --
funcTable' ::
  [(String, [WVal] -> ThrowsError WVal)]
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 7: Flexible type system: Type-checking now takes place in if' and car function definitions. No restrictions on types in funcTable' definitions.

We have introduced Wrangell a novel DSL with familiar Lisp-like syntax which is particularly well suited to data-munging style applications. Wrangell provides convenient utilities and syntax for easily manipulating and transforming data, and outputting to any sort of file given that an implementation is provided.

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). It would also be useful if Wrangell could handle files with inconsistent ordering of columns. Additionally, we would want Wrangell to be able to handle data tables with an unknown number of columns.

As mentioned in our discussion of Wrangell's capabilities for reading and writing csv type files, Wrangell might also be extended to read in and output more file formats.

6. REFERENCES

[1] Write yourself a scheme in 48 hours/adding variables and assignment. https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours/Adding_Variables_and_Assignment, 2014.

[2] J. Goerzen. missingh/src/data/csv.hs. https://github.com/jgoerzen/missingh/blob/master/src/Data/CSV.hs, 2016.

[3] M. Grabmüller. Monad transformers step by step. https://page.mi.fu-berlin.de/scravy/realworldhaskell/materialien/monad-transformers-step-by-step.pdf, 2006.

[4] P. Hudak. Domain specific languages. http://cs448h.stanford.edu/DSEL-Little.pdf, 1997.

[5] P. Hudak, J. Peterson, and J. Fasel. A gentle introduction to haskell: Input/output. https://www.haskell.org/tutorial/io.html, 2000.

[6] G. Sussman and G. Steele. Scheme: An interpreter for extended lambda calculus. https://dspace.mit.edu/bitstream/handle/1721.1/5794/AIM-349.pdf, 1975.