FRY - A Flat File Data Processing Language

Tom DeVoe
tcd2123@columbia.edu

September 29, 2014

1 Introduction

Computers have been used for data analysis since the days of UNIVAC and IBM System/360 and the field is still growing. From Big Iron to Big Data, there will always be a need for methods which enable users to process and transform data sets. There are plenty of languages around which allow you to do this - R and SQL are more tailored towards this functionality, but you can even accomplish this processing even with more general purpose languages like Java or Python.

However, analyzing data sets with pure imperative languages feels awkward to me. There are a lot of times when you are performing actions on groups of records where loops feel heavy-handed. SQL allows you to process data sets nicely without having to use loops, but SQL is only easily used with databases, and is much too verbose for my liking. So, I am setting out with the goal to develop **FRY**[1], a language for querying, processing, and transforming data in flat text files.

2 Language Overview

The broad vision of **FRY** is of a language which will allow for a concise definition of any arbitrarily complex transformations. The way that users can define these transformations succintly is by combining any of the **Primitive Transformations** into **Complex Transformations**, which are a special type of function which takes in a set and returns a transformed version of that set. Combining these Complex Transformations into more elaborate transformations will allow for succint expression of complicated business logic or data analysis. **FRY** will also allow for regular **functions** which can return any of the other Built-in Data types.

The syntax of the language will be a combination of a dynamic and static type system. When defining a **Record** (a data-type on which I will elaborate on in the following section) or a **function** you will need to explicitly define the types of the fields which make it up. Outside of these cases, the type system will be dynamic.

FRY will borrow elements from both functional and imperative programming languages. There are times when processing data that operations need to be repeated and there are times when operations on a data set are much more nicely expressed through functional means. So, a number of different control flow statements will be taken from imperative languages (`for`, `while`,`if`,`elif`), but there will also be functional-like set operations (à la SQL).

2.1 Built-in Types

- `int` - an 8-byte integer value.
- `str` - A text value. Can be a single character or an arbitrarily long string.
- `float` - A double-precision floating-point number
- `bool` - a boolean value, either `true` or `false`
- `Field` - an array of a specific type of element. Every column in a data set is a *field*
- `Record` - a collection of different data types. Can used to map the record layout of a data set, for creation of complex data types. Every row in a data set is represented by a *record*.
- `Set` - A data set which you can perform a number of operations on to enable data transformations. A Set is made up of Records (the rows), Fields (the columns), and a layout which describes the data.

[1] The name is just an homage to a certain television show I am fan of and has no special meaning w.r.t data processing

2.2 Operators

- `( )` - Parentheses are used to indicate the quantity inside should be treated as a Set. Also used when calling functions.
- { } - Brackets can indicate either *(a)* the definition inside is a record or *(b)* when combined with the dot operator on a composite data type, that you are accessing a subset of that data type's elements *(c)* the inside of a function or a control section.
- `.` - The dot operator allows you to access elements of any of the composite data types (Field, Record, or Set)
- `$` - The dollar sign substitutes a variable with it's value. Can be used to dynamically access different fields by name.
- `* / + - ^ < >` - The mathematical operators will all perform their standard behavior on `ints` and `floats`. On `strs`, the `+` sign will serve as the concatenation operator.
- `%` - The percent sign will serve as a wildcard character for string matching.
- `==, !=` - The equivalence operators will compare values by value, not reference.
- `&&, ||` - Double ampersand and double vertical bar represent logical AND and OR, respectively.
- `Read/Write` - These keywords will allow the program to read or write to a file or *stdin/stdout/stderr*
- `if, elif, for, while` - These keywords will control flow, as they do in most imperative programming languages. for loops will support iterating over elements using the `for ... in` syntax used in Python and Bash.

2.3 Primitive Transformations

The meat and potatoes of **FRY** are a few **Primitive Transformations**, which when used in combination allow the programmer to express any arbitrarily complex data transformation. All these transformations return a set. These transformations are:

1. Return a subset of Fields from a set or record

   For some input set of records S with fields $F1, F2, F3$, you can obtain a subset of those fields and assign it to a new set $S2$ like so:

   ```
   # Returns a subset of the fields
   S2 = ( S.{F1,F2} )
   # Returns a field
   values = S.F1
   # Returns all fields with the exception of those listed
   S2_2 = S.{!F1}
   ```

2. Conditionally return a subset of records from a set

   For set $S3$ with fields $F1, F2, F3$, where $F1$ is an `int` or a `float`, you can return all records where $F1 > 15$ and assign it to a new set $S3$

   ```
   S4 = ( S3 if F1 > 15 )
   ```

3. Sort a set on one or more fields, collating based on the system's *locale*

   ```
   S5 = ( Sort S4 on F1, F2 )
   ```

4. Join two sets together on one or more fields You can choose to return matched and/or unmatched records from either side (the keywords `UMR`, `M`, and `UML` describe which records to return)

   ```
   S6 = ( Join S1.F1 == S2.F1, S1.F2 == S2.F2  ret UMR,M)
   ```

5. Concatenate two record sets with the same format

   ```
   S7 = ( S1, S2 )
   ```

3 Typical Use Cases

The aim for **FRY** is to be able to accomplish any data processing necessary. This includes:

- **Data Validation** - Flagging, fixing, or removing bad data from a predefined data set
- **Extracting Relevant Data** - Extracting data relevant to certain queries (e.x. all Users who are under the age of 23 or all payments made from the state of Missouri)
- **Merging Data Sets from Different Sources** - Integrating data sets from multiple sources, potentially all with different file layouts
- **Tracking Data Changes** - Tracking and recording records which have changed, have been deleted, etc. One common use case of this is a CDC, illustrated in the example.
- **Analyzing Data** - Statistically analyzing data to gain further insight

These examples are often combined to perform more intricate processing. This is just a subset of the ways that **FRY** can be used to process data.

4 Example Program: CDC

The following example programs implement a CDC. CDCs are a common process used in data-warehoues to capture and track changes in frequently updated data. In this example, the program takes in a file with the current (updated) records and a file with the previous records. Any record in the previous file and not the current file was deleted, any record not in the previous file and in the current file is new, and any matching record with different field values in the previous file and current file is changed. I have shown how one could in **FRY** two different ways:

- The case which only needs to handle a specific data set
- A general case where there I define a complex transformation which can perform a CDC on any data set

Specific Case

```
# This defines the layout of the data and reads in the two input files

layout =  {int key, str fname, str lname, str addr, delim ","}
previous_set = ( Read "/home/tdevoe/previous.txt", layout )
current_set = ( Read "/home/tdevoe/current.txt", layout)

# Get the deleted, new, and matched records by joining the two files on the key.
# Fields with a common name are accessed by referencing the
# side they came from (R or L)

deleted_set = ( ( Join previous_set.key=current_set.key ret UML).{key, L.fname, L.
    lname, L.addr} )
new_set = ( ( Join previous_set.key=current_set.key ret UMR ).{key, R.fname, R.lname
    , R,addr} )
matched_set = ( Join previous_set.key=current_set.key )

# Need to check the records in matched_set to find out if they are changed

changed_set = ( matched_set if L.fname != R.fname or L.lname != R.lname or L.addr !=
     R.addr )

# write each of the changed/deleted/new records to their correspond output file

( Write "/home/tdevoe/deleted_records.txt", deleted_set, "append")
( Write "/home/tdevoe/new_records.txt", new_set, "append" )
( Write "/home/tdevoe/changed_records.txt", changed_set, "append")
```

General Case

```
## Definition of a generalized CDC transformation
## Takes in the two sets being processed and the name
## of the key field
trans CDC (Set current_set, Set previous_set, str key) {

        # Get the deleted, new, and matched records by
        # joining the two files on the key.
        # Fields with a common name are accessed by referencing the
        # side they came from (R or L)

        deleted_set = ( Join previous_set.$key=current_set.$key ret UML)
        new_set = ( Join previous_set.$key=current_set.$key ret UMR )

        # When fields are joined with the same name
        # The fields are renamed so the field from the left is L_$fieldname
        # and the right is R_$fieldname
        matched_set = ( Join previous_set.$key=current_set.$key )

        # This gets the layout of the original data
        orig_fields = current_set.layout.{!$key}


        # The ":" indicates that the value before it should be set to
        # its value after the code following the ":" is completed
        changed_set = ( matched_set if isChanged:
                        isChanged = true
                        for fld_name in orig_fields {
                                isChanged = isChanged && { L_${fld_name} == R_${
                                    fld_name}
                                }
                        }
                )

        # The function addValue will be a transformation implemented
        # in the standard library. It's purpose is to take a set and append
        # a value to every record in the set and adding a field
        # with the third argument's name to the layout

        ret ( (addValue(deleted_set, "D", "status")), (addValue(new_set, "N", "
            status")), (addValue(changed_set, "C", "status")) )

}


# This defines the layout of the data and reads in the two input files

data_layout =  {int key, str fname, str lname, str addr, delim ","}
previous_set = ( Read "/home/tdevoe/previous.txt", data_layout )
current_set = ( Read "/home/tdevoe/current.txt", data_layout)

processed_data = ( CDC(current_set, previous_set, "key") )

# write each of the changed/deleted/new records to their correspond output file

( Write "/home/tdevoe/deleted_records.txt", ( processed_set if status == "D" ) , "
    append")
```

```
( Write "/home/tdevoe/new_records.txt", ( processed_set if status == "N" ), "append"
    )
( Write "/home/tdevoe/changed_records.txt", ( processed_set if status == "C" ), "
   append")
```