Final

Vision

Trauma is a prototype language for a concept called differential dataflow outlined in the following paper
https://link-springer-com.proxy.library.cornell.edu/chapter/10.1007/978-3-662-46678-0_5.

The goal of this project is to be able to create a programming language that is able to do incremental iterative calculations faster. For example, consider the algorithm to find all the connected components of a graph: label all nodes and iterate through all nodes making sure that we relabel each node to be the minimum of its neighbors. If we change the graph such that one edge is deleted, the naive method to apply the algorithm on the new graph. Trauma fixes this problem by using results from computing the connected components of the original graph to solve the new graph.

In order to do this, we use collections and traces. Collections are simply just multisets. Traces are indexed multisets or indexed other traces. Traces store differences rather than the actual version of the data. Differential operators are able to use these differences to efficiently calculate new collections.

To illustrate this idea, we can consider an arbitrary operator f. To get the difference between f(version4) and f(version5) where f(versionx) was calculated for x < 5, we have to do f(sum our differences up to version5) - sum of f(versionx) from x = 0 to 4. Now our problem here is that we are just doing f(version5) and no efficiencies are introduced. In fact, we have to calculate even more because we need to take a sum to get version5 in the first place. The efficiencies come in when we modify this algorithm a little for specific operators. For example, data parallel operators have the liberty of only considering the keys in our collection that change.

Loops are interesting because they extend the dimensionality of the index of the trace we are working with. We then get a trace that is partially ordered. Differential dataflow proves that we can still take differences that are before a version in this partial order and apply the same idea as we explained above.

Summary of Progress

At this point I have a much better understanding of the subject. This week I focused on…

1) I redid my **Trace** and **Collection** modules to generalize so that Traces may be recursively defined to be a structure where we index on other Traces.

2) I added a new **values** type to my interpreter to indicate values and cleaned up some unused expressions to make things more clear while polishing old ones.

Status

A pdf of the expressions and their operation semantics are under the file `semantics.pdf` in the repo. We were not able to complete **iter**.

Activity Breakdown

1) **Traces** and **Collections** Modules
    a) Increased the complexity of **Traces** and **Collections** by adding in that **Traces** be a recursive type
    b) Traces can now have multiple dimensions (getting it ready for **loop** expressions)
    c) Created a formal OUnit2 testing suite
    d) Implement **distinct** differential operator
2) Reorganizing project
    a) Created and implemented well defined syntax and rules
    b) Eliminated older unused expressions

Productivity Analysis

A lot of time was thinking of the algorithms for traversing the **Traces** and **Collections** properly when working with them.

Grade

I give myself an A for this phase. I think I did really well for this phase because of how much better I understand how everything fits together. A lot of the struggling through this was because the implementation was being done on OCaml where lists are harder to work with especially when they are nested. I was able to get through it however and at least have a proof of concept of the underlying math behind the differential dataflow paper I linked above in my unit tests. Although I was able to implement the one dimensional trace case in my language, I unfortunately wasn't able to show multiple dimensions because I couldn't implement the loop. I was getting confused when trying to implement the feedback aspect of the loop in my **Trace** module. I was also able to implement one differential operator that used the general algorithm I provide in the introduction. I was also better able to define my semantics and rules.

Next Steps

1) Redo the language in an imperative language. This would allow me to more easily work with lists for my traces. Furthermore my collections would be represented as hashmaps and I will be able to index by keys to open room for efficiency with data parallel

operators. Currently my implementation is just a proof of concept for the underlying theory but does not introduce the efficiencies we want.

2) Implement loops. I believe that working with an imperative language where lists are much more manageable will help me see ways to implement specifically the feedback component of loops more easily.