Incremental Computing in Haskell
CS240H Project Report

Jiyue Wang and Kaixi Ruan

March 18, 2016

1 Introduction

In this project, we implement a Haskell library for incremental computations. Incremental computing, or self-adjusting computing, is the idea of tracking data and control dependencies in order to selectively re-evaluate only parts of the computations given some changed inputs.

A simple and well-known example of incremental computing is the spreadsheet. In a spreadsheet, a cell contains either some simple data, or an equation describing how the result should be derived from values in other cells. It is critical for a spreadsheet to avoid re-evaluating all the cells when only one cell has changed value.

Jane Street recently released their open-source library written in OCaml named *Incremental*[4], which concretizes this idea. It is natural for us to inquire whether a similar library exists for Haskell.

It turns out that there are already some existing libraries in Haskell related to incremental computing: *Adaptive*[2] is one directly related to Incremental Computing, but the library was based on some theoretical results int the year 2002 and the theory has evolved since then. There are also many Functional Reactive Programming libraries in Haskell, such as *Reactive-banana*[1]. However, this library is mostly focused on GUI applications rather than computations.

In the following sections, we describe how a library like Jane Street's Incremental is implemented using Haskell.

2 Incremental in a nutshell

2.1 Incremental DAG

The key idea behind Incremental Computing is a Directed Acyclic Graph(DAG).

In an Incremental DAG, a node represents an incremental value in the computation: it can be a simple variable input, or it can depend on some other incremental values.

An edge exists from node `n` to node `m` if the value of node `m` depends on the value of node `n`. We say that node `m` is a parent of node `n`.

2.2 Demo

Necessary Node

Variable user could create variables which they could later change value

Operation like map/bind/arrayfold/...

Observer use observer to observe some node/make it necessary

Stabilize after building the graph/change the value, use stabilize to ...

Garbage Collection

3 Implementation

Representing a dynamic graph in Haskell is not as straightforward as tree. For efficiency consideration, we decided to give up purity and use `IORef`. The good news is that we only need to keep one copy of each node, while on the other hand, most of the manipulation will involve `IO` monad.

3.1 Node

`Node a` represents a node in the DAG. Each node needs to maintain a list of fields which might be updated during the stabilization. To avoid copying large record, we organize the fields in a hierarchical structure and use `Lens.Simple` to get easy access and mutation.

Listing 1: Node

```haskell
data Node a = Node {
    _kind     :: Kind a
```

```
  , _value    :: ValueInfo a
  , _edges    :: Edges
  }

data Kind a =
    forall b. Eq b => ArrayFold ...
  | forall b. Eq b => Bind {
        func  :: b -> StateIO (NodeRef a)
      , lhs   :: NodeRef b
      , rhs   :: Maybe (NodeRef a)
      , nodesCreatedInScope :: [PackedNode]
      }
  | Const a
  | Variable {
        mvalue  :: a
      , setAt   :: StabilizationNum
      , valueSetDuringStb :: !(Maybe a)
      }
  | forall b. Eq b => Map (b -> a) (NodeRef b)
  ...

data Edges = Edges {
    _parents   :: Set PackedNode
  , _obsOnNode :: Set ObsID
  }

data Scope = Top
           | forall a. Eq a => Bound (NodeRef a)

data NodeRef a = Ref (IORef (Node a))
                     !Unique -- node id
                     !Scope  -- scope created in

data PackedNode = forall a. Eq a =>
                  PackedNode (NodeRef a)
```

kind could be `Variable`, `Map`, `Bind`, etc., which represents the type of the node. It also keep references to all possible children when it is first created. However, the child node does not necessarily has an edge to its parents. The child-to-parent edge is added (from parent) only when the parent becomes necessary and it is removed once the parent is unnecessary.

value not only contains the current node value but also contains some extra information to help decide whether the value is stale.

edges stores the topological information of the graph. It contains references to parent nodes as well as observers watching the current node.

Unique gives a unique identifier for each `NodeRef` that helps to compare nodes of different types without dereferencing the `IORef (Node a)`.

Scope indicates the scope in wich the node is created. A user could introduce a new node in the 'global' scope (`Top`), or on the RHS of a `Bind` node. This is useful when the recomputation involves a `Bind` node. As both **id** and **scope** is immutable during the lifetime of a node, we could keep them outside `Node a`, thus saving one layer of indirectness.

PackedNode is a convenience wrapper over nodes of different types. This allows us to store heterogeneous parent/child nodes.

3.2 Observer

Users can only change the value of a `Variable` node, but they can read other kind of node by adding an `Observer` to the node in interest. An `InUse` oberver makes the observed node necessary. Remember that only necessary nodes will appear in the DAG and update during stabilization.

Listing 2: Observer

```
newtype Observer a = Obs ObsID

type ObsID = Unique

data InterObserver a = InterObs {
    _obsID  :: !ObsID
  , _state  :: !ObsState
  , _observing :: !(NodeRef a)
  }

data ObsState = Created
              | InUse
              | Disallowed
              | Unlinked

data PackedObs = forall a. Eq a =>
                 PackObs (InterObserver a)
```

3.3 State

We need something like `State` monad to keep track of the DAG and observers. To incorporate `IO` monad as well, we use the monad transformer `StateT` to stack them into a new monad `StateIO`.

Listing 3: State

```
type StateIO a = StateT StateInfo IO a

data StateInfo = StateInfo {
    _info           :: StatusInfo
  , _recHeap        :: Set PackedNode
  , _observer       :: ObserverInfo
  , _varSetDuringStb :: [PackedVar]
  }
```

info keeps track of status related information, including whether the program is during a stabilization, stabilization number, current scope and debug information, etc.

recHeap is a somewhat misnamed field. It is used be a minimum heap which stores the necessary nodes that needs to recompute during next stabilization. Later, we use DFS-based topological sorting to update nodes and this field becomes a set of root nodes for DFS.

observer is a map of observer ID to instances. Currently, we use a standard map which based on size balanced binary trees. It could be easily replaced by other containers like `IntMap` to improve performance.

varSetDuringStb is a list of variables set during stabilization, used in asynchronous stabilization.

3.4 Stabilization

3.4.1 `Bind` node

After the user adds observers or make changes to variables, they need to call `stabilize` or `stabilizeAsync` to trigger the recomputation. The algorithm is a little complicated because of the `Bind` node.

For a static graph, the algorithm is straightforward. First, it starts DFS from the nodes in `recHeap` and gets a list of nodes. It then recomputes nodes in the list sequentially and updates all the necessary nodes.

This algorithm will not work with a `Bind` node (see Listing[4]) that generates the graph on the fly. To deal with `Bind` node, we do the following modification (see Algorithm[1]). Note that the 'else' part actually solve three possible cases and we check the stabilization number before recomputing the node to avoid duplicated work.

a. `rhs` is `Nothing`.

b. Only `lhs` changes.

c. Both `lhs` and `rhs` changes.

Listing 4: Bind node

```
data Kind a = ...
  | forall b. Eq b => Bind {
     func  :: b -> StateIO (NodeRef a)
   , lhs   :: NodeRef b
   , rhs   :: Maybe (NodeRef a)
   , nodesCreatedInScope :: [PackedNode]
   }
  ...
```

```
if $stbNum(LHS) < stbNum(Current)$ then
    (LHS does not change);
    copy the value of RHS node;
else
    (Regenerate RHS);
    run func;
    recompute RHS nodes recursively;
    update rhs if necessary;
end
```

Algorithm 1: Recompute `Bind` node

3.4.2 Asynchronous Stabilization

Considering that a large graph may take time to recompute, we provide `stabilizeAsync`, `waitForStb` and a helper function `amStabilizing` to forward the recomputation to another thread. During the stabilization, the user is allowed to create and modify nodes as well as observers. However, all the actions taking place during current stabilization will not take effect until the next stabilization. Check out Listing[5] for an example.

Listing 5: Asynchrnous Stabilization

```
exampleAsync :: StateIO ()
exampleAsync = do
  v1 <- var (5 :: Int)
  b1 <- (const True) >>=|
        (\_ -> expensiveWork 1000 >> return
             (watch v1))

  ob <- observe b1
  stabilizeAsync
  ob' <- observe b1

  waitForStb
  printObs ob
  -- printObs ob' -- should got exception
  stabilize
  printObs ob'

expensiveWork :: Int -> StateIO ()
expensiveWork n = lift (putStr $ (take n $
    repeat '.'))
```

This example also shows that it is user's responsibility to make sure that the function fed into `Bind` node have no other side effect except for introducing new nodes or observers. It is easy to crash the system with inappropriate `IO` actions and thus we are trying to restrict misuse on the type level to improve safety.

3.5 Exception Handling

Runtime exception is inevitable as incremental provides users with the flexibility to construct the DAG on the fly. However, the exception handling fuctions in `Control.Exception.Base` only work with `IO` monad, not monad transformers. Exception handling in monad transformers could be tricky (see [3]). Luckily, `exceptions` package gives a clean solution in our case.

4 Future Work

4.1 Testing

Currently, we don't have enough time to write thorough test cases for the library. As most of the function has `StateIO` in their signature, we plan to use `Test.QuickCheck.Monadic` help us.

4.2 Complete the Functionality

So far, our library provides sufficient elementary functions like `map`, `map2`, `bind`, etc. To make the user's life easier, we are adding more functionality like `arrayFold`, `ifElseThen`, `sum`, etc. In addition, it will be nice to log the current state of graph in local file system and recover/resume the computation. Also, we are completing the exception handling mechanism to help reduce runtime exception caused by misuse.

4.3 Improve the Algorithm

Our current algorithm keeps a global state to keep track of the graph and uses a DFS-based topological sorting algorithm to do recomputation. This approach is apparently not parallel.

References

[1] H. Apfelmus. Reactive-banana. `https://wiki.haskell.org/Reactive-banana`. 2016-01-04.

[2] M. Carlsson. Adaptive. `https://hackage.haskell.org/package/Adaptive`. 2013-01-28.

[3] M. Snoyman. Exceptions and monad transformers. `https://www.schoolofhaskell.com/user/snoyberg/general-haskell/exceptions/exceptions-and-monad-transformers`. 2013-08-22.

[4] J. Street. incremental. `https://github.com/janestreet/incremental/blob/master/src/incremental_intf.ml`. 2015-08-17.