Incremental Computing in Haskell
CS240H Project Report

Jiyue WANG and Kaixi RUAN

March 17, 2016

1 Introduction

2 Incremental in a nutshell

2.1 Incremental DAG

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

```
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 be 'safe'. It is easy to crash the system with inappropriate `IO` actions.

4 Future Work

4.1 Testing

4.2 Exception Handling

4.3 Add Functionality

4.4 Improve the Algorithm