```
%include polycode.fmt
%format ^* = "\ltimes"
%format *^ = "\rtimes"

\subsection{Description of the quantum stack}\label{subsec:quantumstackdescription}

%if false
\begin{code}
{-#OPTIONS_GHC -fno-warn-orphans #-}

module QSM.QuantumStack.QSRotation
    (rotateup,preconditionQdataStructs,rotateInOrder,
             discard, (+^+), (-^-), (+/+), (-/-), qsNegate,
             pullSingle, pullup, fixDiagonal
    )
 where
import QSM.QuantumStack.QSDefinition
import QSM.QuantumStack.QSManipulation
import QSM.MachineErrors
import Data.ClassComp
import Data.ClassicalData
import QSM.BasicData
import Data.Tuples
import Data.List
import Utility.Extras


\end{code}
%endif

The function |rotateup| works in conjunction with the |pullup| and |descendMap|
functions to rotate a node
 to the top of the stack. The first argument
is the address of the node
 to bring up. The second is the |QuantumStack| to
rotate.

The function works by doing a recursive check and descent.
There are two cases.
\begin{description}
\item[Already at top] --- Just return the current stack.
\item[Somewhere else] --- Use |descendMap| to rotate the desired
node to the top of each of the sub-stacks, then apply |pullup|.
\end{description}

{
\begin{singlespace}
\begin{code}

rotateInOrder :: (Quantum b) => [StackAddress] -> QuantumStack b ->
                 QuantumStack b
rotateInOrder [] q = q
rotateInOrder (nm:nms) q = rotateInOrder nms $ rotateup nm q

rotateup ::  (Quantum b) => StackAddress -> QuantumStack b ->
            QuantumStack b
rotateup s stack
         | isStackLeaf stack = stack
         | s == address stack
             = stack
         | otherwise
```

```
             = --trimStack $
               pullup s $
               descendMap (rotateup s) stack
\end{code}
\end{singlespace}}

In order to pull up nodes, the  function |pullSingle|
is defined to work on nodes \emph{with a single sub-node}.
This simplifies the rotation issues when nodes have
constructors with different bound sub-nodes.


{\begin{singlespace}
\begin{code}
pullSingle :: (Quantum b) =>
              StackAddress ->
              QuantumStack b ->
              QuantumStack b
pullSingle _ qs@(QuantumStack _ _ _ StackZero) = qs
pullSingle _ qs@(QuantumStack _ _ _ (StackValue _)) = qs
\end{code}
\end{singlespace}}

The details of the pull up code are elided.

The function |pullSingle|
handles the case of the
search address being either on top or the
second element of the |QuantumStack|. As with |rotateup|, if
the target node is already on top,
no change is made. In the case where the target node
 is the next element on the stack,
an exchange is done to bring the second element to
the top of the stack. This is achieved by
building a new stack with the second element at
the top and the first element
at the first set of sub-stacks. The sub-stacks
 are re-ordered so that their
"tree path" remains the same.
The "tree path" can be thought of as the path
needed to descend the sub-stacks.

%if codeOnly || showSupportFunctionDetail
\begin{code}
pullSingle s qs
    | s == address qs = qs
    | 1 == length (subStacks qs) =
        let qs2 = qHead "pullSingle " $ subStacks qs -- only one
            tsubsubs = map (: []) $ subStacks qs2
            remapped = map (\x -> qs{onDiagonal = False, subStacks = x}) tsubsubs
        in  qs2{onDiagonal = onDiagonal qs, subStacks = remapped}

pullSingle s qs = error " Pulling missed out"


fixDiagonal :: (Quantum b) => Bool -> QuantumStack b -> QuantumStack b
fixDiagonal a qs
    = let ss = subStacks qs
      in case  descriptor qs  of
               StackQubit _ -> if a then diagonalize qs
                               else undiagonalize qs
```

```
                _               -> qs{onDiagonal = a,
                                      subStacks = map (fixDiagonal a) ss}

diagonalize :: (Quantum b) => QuantumStack b -> QuantumStack b
diagonalize qs@(QuantumStack a _ _ (StackQubit _))
    = QuantumStack a True [fixDiagonal True (zz qs), fixDiagonal False (zo qs), fixDiagonal Tr
ue (oo qs)] (StackQubit diagQbs)

undiagonalize :: (Quantum b) => QuantumStack b -> QuantumStack b
undiagonalize  qs@(QuantumStack a _ _ (StackQubit _))
    = QuantumStack a False
           (map (fixDiagonal False) (qvalues qs))
           (StackQubit fullQbs)




\end{code}
%endif


The function |pullup| breaks the input |QuantumStack| into
a list, each of which is a |QuantumStack| with a single sub-node.
|pullSingle| is applied to this list and the results are merged.


{\begin{singlespace}
\begin{code}

pullup :: (Quantum b) =>
          StackAddress ->
          QuantumStack b ->
          QuantumStack b
pullup s qs = fixDiagonal (onDiagonal qs) $
              foldr (+^+) zerostack $ map (pullSingle s) $ doBreak qs

\end{code}
\end{singlespace}
}




The function |preconditionQdataStructs| is used when doing a transform on a quantum stack. It
will
rotate up the requisite number of \qubits{} for the transform.

FIXME UNUSED!!!!

{\begin{singlespace}
\begin{code}

preconditionQdataStructs ::  StackAddress -> Int -> QuantumStack b ->
                             ((QuantumStack b -> QuantumStack b),QuantumStack b)
preconditionQdataStructs _ _ qs                = (id,qs)
--preconditionQdataStructs Nothing n q@(StackCons _ _)  =
--    let (oldnms, upnms) = getUpAddresses n q
--        q' = foldr (flip rotateup) q $ reverse upnms
--    in (\qs -> (rotateInOrder $ reverse oldnms ) qs, q')
```

```
--preconditionQdataStructs _ _ qs                          = (id,qs)
\end{code}
\end{singlespace}}

The function |getUpAddresses| is used only by |preconditionQdataStructs| to determine
the addresses of both the data nodes that are going to possibly be
rotated down and the addresses of the \qubit{}s to rotate up. The
function |getUpAddresses| will only return
the required number of \qubit{} addresses.

It uses the subordinate functions |getQubits| which returns
a list of the bound variables which are \qubits{}  and |getDstacks|
which returns a list of pairs of addresses of other data nodes and \qubits{}
at lower levels in the quantum stack. These subordinate functions
are explained more fully below.



{\begin{singlespace}

getUpAddresses ::(Quantum b) => Int -> QuantumStack b ->
             ([StackAddress],[StackAddress])
getUpAddresses n qs
    | n <= 0 = ([],[])
getUpAddresses n (QuantumStack nm _ [sstack] (StackData [(cons,addrs)]))
       = let qbnms = getQubits addrs sstack
             dstacks = getDStacks addrs sstack
             m = n - (length qbnms)
             moreaddresses = foldr (\(x,y) (a,b) -> (x ++ a, y++b)) ([],[]) $
                         (map  $ getUpAddresses m) dstacks
         in (nm:fst moreaddresses,
               take n $ qbnms ++ (snd moreaddresses))
getUpAddresses _ _  =
    error "Indeterminate qubit order for transform"

\end{singlespace}}

The function |getDStacks| first filters its first argument (a list of
|StackAddress|es) by determining which of them are data nodes. These
are then paired with the current quantum stack and in each case, that node
is rotated up to the top of the quantum stack. This is then
suitable for further recursion by |getUpAddresses|

The filtering function |isDtype| returns true if the top node is
a data node with the correct address, otherwise it recurses down the quantum stack.


{\begin{singlespace}


getDStacks ::Quantum b => [StackAddress] -> QuantumStack b ->
            [QuantumStack b]
getDStacks nms q =
    (map (uncurry rotateup)) $ getDtypesAddrsAndQs nms q

getDtypesAddrsAndQs ::Quantum b => [StackAddress] ->
                  QuantumStack b ->
                  [(StackAddress, QuantumStack b)]
getDtypesAddrsAndQs addrs q =
    zip (filter (flip isDtype q) addrs) $ repeat q
```

```
\end{singlespace}}

{\begin{singlespace}
\begin{code}


discard :: (Quantum b) =>  StackAddress ->
           QuantumStack b ->
           QuantumStack b
discard addr  = byAddress addr discardit


discardit :: (Quantum b) =>
           QuantumStack b ->
           QuantumStack b
discardit stk =
  if onDiagonal stk then
     case descriptor stk of
        StackClassical _ ->
          foldr (+/+) zerostack $ subStacks stk
        StackQubit _  ->
          zz stk +/+ oo stk
        StackData dvals ->
          let addrs = map snd dvals
              stacks' = map (discardEach addrs) $ subStacks stk
          in foldl' (+/+) zerostack stacks'
        _ -> stk
  else
     case descriptor stk of
        StackClassical _ ->
          foldr (+^+) zerostack $ subStacks stk
        StackQubit _  ->
          zz stk +^+ oo stk
        StackData dvals ->
          let addrs = map snd dvals
              stacks' = map (discardEach addrs) $ subStacks stk
          in foldl' (+^+) zerostack stacks'
        _ -> stk


discardEach :: (Quantum b) =>   [[StackAddress]] ->  QuantumStack b ->  QuantumStack b
discardEach  = foldl' (.) id . map discard . concat

\end{code}
%endif
%TODO The second level one is interesting in that there may be
%different addressess bound under the same constructor. I'm not sure what
%is the correct thing to do in that case. Is it as simple as just renaming
%the conflicting
%ones. Is there a deterministic way of doing that?

%Or does that imply a different structure is really required, i.e. a map from
% (|Constructor|, [StackAddress]) pairs to the sub stacks. My gut likes the second
%better. i.e., you can have a list that is Nil OR Cons ('1', Nil) OR
%Cons ('0', (Cons '1', Nil)). I'll continue for now with out changing that.
%
%For example how would you rename to Nil and Cons ('1', Nil) being the same thing -
%doesn't make sense to me.

%For now, Take head of addresses
```

```
The function |addable| ensures the two stacks are
compatible \emph{at the top level}. Substacks may be in a different order.

Two functions, $+^+$ and $+/+$ are defined where the first expects
to deliver a non-diagonal result and the second a diagonal result. The
arguments to either can be diagonal or non-diagonal.

%if codeOnly || showClassDerivations
\begin{code}

addable ::  (Num b) => QuantumStack b ->  QuantumStack b -> Bool
addable s1 s2 = isStackZero s1 ||
                isStackZero s2 ||
                address s1 == address s2


(+^+) :: (Quantum b) => QuantumStack b -> QuantumStack b ->
         QuantumStack b
(+^+) qs@(QuantumStack _ _ _ StackZero) a
    | isStackQubit a
        = let qvs = qvalues a
          in setQvalues qvs (a{onDiagonal = False})
    | otherwise  = a
(+^+) a qs@(QuantumStack _ _ _ StackZero)
    | isStackQubit a
        = let qvs = qvalues a
          in setQvalues qvs (a{onDiagonal = False})
    | otherwise  = a
(+^+) qs1@(QuantumStack _ _ _ (StackValue a))
       qs2@(QuantumStack _ _ _(StackValue b))
           =  qs1{onDiagonal = False, descriptor = StackValue (a+b)}

(+^+) qs1@(QuantumStack a1 _ _ (StackQubit _)) qs2@(QuantumStack a2 _ _ (StackQubit _))
    | a1 == a2
        = setQvalues (zipWith (+^+) (qvalues qs1) (qvalues qs2)) (qs1{onDiagonal = False})

(+^+) (QuantumStack a1 _ s1 (StackClassical cvs1)) (QuantumStack a2 _ s2 (StackClassical cvs2)
)
    | a1 == a2 = let (rcvs,rs) = unzip $ stackUnionWith (+^+)
                                 (zip cvs1 s1) (zip cvs2 s2)
                 in (QuantumStack a1 False rs (StackClassical rcvs))
(+^+) (QuantumStack a1 _ s1 (StackData dv1))
          (QuantumStack a2 _ s2 (StackData dv2))
    | a1 == a2 = let (rdv,rs) = unzip $ stackUnionWith (+^+)
                                (zip dv1 s1) (zip dv2 s2)
                 in (QuantumStack a1 False rs
                     (StackData rdv))
(+^+) stk1 stk2 =  let  nms1 = address stk1
                        stk2' = rotateup nms1 stk2
                   in  if addable stk1 stk2' then stk1 +^+ stk2'
                       else  let  stk1' = trimStack Nothing stk1
                                  nm1'  = address stk1'
                                  stk2'' = trimStack Nothing $ rotateup nm1' $ trimStack Nothi
ng stk2
                             in  if addable stk1' stk2'' then stk1' +^+ stk2''
                                 else error $ badqsAdd (show stk1) (show stk2)


(-^-) :: (Quantum b) => QuantumStack b -> QuantumStack b -> QuantumStack b
(-^-) a b = a +^+ qsNegate b
```

```
(+/+) :: (Quantum b) => QuantumStack b -> QuantumStack b ->
        QuantumStack b
(+/+) qs@(QuantumStack _ _ _ StackZero) a
    | isStackQubit a
      = setQvalues [zz a, zo a, oo a] (a{onDiagonal = True})
    | otherwise  = a
(+/+) a qs@(QuantumStack _ _ _ StackZero)
    | isStackQubit a
      = setQvalues [zz a, zo a, oo a] (a{onDiagonal = True})
    | otherwise  = a
(+/+) qs1@(QuantumStack _ _ _ (StackValue a))
       qs2@(QuantumStack _ _ _(StackValue b))
          =  qs1{onDiagonal = True, descriptor = StackValue (a+b)}

(+/+) qs1@(QuantumStack a1 _ _ (StackQubit _)) qs2@(QuantumStack a2 _ _ (StackQubit _))
    | a1 == a2
      = setQvalues [zz qs1 +/+ zz qs2,
                    zo qs1 +^+ zo qs2,
                    oo qs1 +/+ oo qs2] (qs1{onDiagonal = True})

(+/+) (QuantumStack a1 _ s1 (StackClassical cvs1)) (QuantumStack a2 _ s2 (StackClassical cvs2)
)
    | a1 == a2 = let (rcvs,rs) = unzip $ stackUnionWith (+/+)
                                 (zip cvs1 s1) (zip cvs2 s2)
                 in (QuantumStack a1 True rs (StackClassical rcvs))
(+/+) (QuantumStack a1 _ s1 (StackData dv1))
         (QuantumStack a2 _ s2 (StackData dv2))
    | a1 == a2 = let (rdv,rs) = unzip $ stackUnionWith (+/+)
                                (zip dv1 s1) (zip dv2 s2)
                 in (QuantumStack a1 True rs
                     (StackData rdv))
(+/+) stk1 stk2 =  let  nms1 = address stk1
                        stk2' = rotateup nms1 stk2
                   in  if addable stk1 stk2' then stk1 +/+ stk2'
                       else  let  stk1' = trimStack Nothing stk1
                                  nm1'  = address stk1'
                                  stk2'' = trimStack Nothing $ rotateup nm1' $ trimStack Nothi
ng stk2
                             in  if addable stk1' stk2'' then stk1' +/+ stk2''
                                 else error $ badqsAdd (show stk1) (show stk2)


(-/-) :: (Quantum b) => QuantumStack b -> QuantumStack b -> QuantumStack b
(-/-) a b = a +/+ qsNegate b

qsNegate :: (Quantum b) =>  QuantumStack b -> QuantumStack b
qsNegate qs = case   descriptor qs of
                     StackValue a -> qs{descriptor =  StackValue (negate a)}
                     _             -> let sns = map qsNegate $ subStacks qs
                                      in qs{subStacks = sns}

\end{code}
%endif
```