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

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

%if false
\begin{code}

module QSM.QuantumStack.QSDefinition
    (val,setAddress,secondAddress,qApply,
     QuantumStack(..),
     StackDescriptor(..),
     qAccessor,cjgt,cjgtTranspose,
     zz, zo, oz, oo,(=~=),
     isStackClassical,isStackQubit,isStackData, isStackValue, isStackZero,
     isStackLeaf, isDiagQubit,
     zerostack,zerovaluedescriptor,
     eraseZeroes, isDtype,getQubits,addresses,
     qvalues,qvaluesDiag, setQvalues,
     setSubStacks,trimSubStacks,
     branchCount,allAddresses,stackUnionWith,
     trace,
     dropzsQ,
     assertQSCorrect
    )
 where
import Data.ClassComp
import QSM.BasicData
import Data.ClassicalData
import Data.Tuples
import Data.List
import Text.Show
import Utility.Extras

\end{code}
%endif

The quantum stack holds both \emph{quantum} and
\emph{probabilistic}
data in the quantum stack  machine.



The quantum stack is a tree with a variable
number of branches at
each level, descriptive data at each level
signifying what kind of node it is and data at the leaves.
New nodes are
always added to the top
of the stack. Manipulations of data at the
top are performed by
applying the proper linear combinations to
the sub-stacks. (For more
detail see |apply|).

Each node consists of a name, a list of sub-stacks, a
"on-diagonal" Bool and
a descriptor.
There are  five
different descriptors for |QuantumStack| elements:
```

```
\begin{description}
\item{|StackZero|} Explicit laziness used when
adding nodes to the
stack. With this case, the sub-stack list is empty.

\item{|StackValue b|} The leaves of the stack
where the entries of the
series of density matrices are stored. This
also serves as a starting
point for a stack, with the value point $b$
 being  $1.0$. The
corresponding sub-stack list is empty.
\item{|StackClassical [ClassicalData]|} A probablistic
classical value. The classical data list will have the
same length as the list of sub-stacks.
\item{|StackData [(Constructor,[StackAddress])]|} A probablistic
algebraic data type. The list of items will have the
same length as the list of sub-stacks. Each pair will consist
of the actual constructor together with a list of |StackAddress|es
that are used by that constructor.

\item{|StackQubit |} A
\qubit{} entry, which will mean there will be either
three or four substacks. If this is a diagonal entry, only
the $00,01$ and $11$ entries are stored as the $10$ entry
 is the conjugate transpose of
the $01$ entry. For non-diagonal entries, all
four substacks are stored.

Remove |Quantum b| constraint on data.
\end{description}
{\begin{figure}[htbp]
\begin{singlespace}
\begin{code}

data  QuantumStack b
    = QuantumStack {address :: StackAddress,
                    onDiagonal :: Bool,
                    subStacks :: [QuantumStack b],
                    descriptor :: StackDescriptor b}

data  StackDescriptor b
    = StackZero  |
      StackValue ! b |
      StackClassical [ClassicalData] |
      StackQubit [(Basis,Basis)]|
      StackData [(Constructor,[StackAddress])]
                deriving Show

(=~=) :: StackDescriptor b -> StackDescriptor b -> Bool
(=~=) StackZero StackZero = True
(=~=) StackZero (StackValue _) = True
(=~=) (StackValue _) StackZero = True
(=~=) (StackValue _) (StackValue _) = True
(=~=) (StackClassical _) (StackClassical _) = True
(=~=) (StackQubit _) (StackQubit _) = True
(=~=) (StackData _) (StackData _) = True

descLength :: StackDescriptor b -> Int
descLength StackZero = 0
descLength (StackValue _) = 0
```

```
descLength (StackClassical cs) = length cs
descLength (StackQubit bs) = length bs
descLength (StackData ds) = length ds


assertQSCorrect :: QuantumStack b -> Bool
assertQSCorrect _ = True
--assertQSCorrect q
--    = case subStacks q of
--       ss -> case  (descLength (descriptor q)) == (length ss) of
--               True -> and $ map assertQSCorrect ss
--               False -> error $ "Missmatch " ++ (show $ descriptor q) ++
--                           " for q="++(show q)


instance (Show b) => Show (QuantumStack b) where
    showsPrec _ (QuantumStack  _ _ _ StackZero )         = shows 0
    showsPrec _ (QuantumStack  _ di _ (StackValue d))     = shows di . shows d
    showsPrec _ (QuantumStack s di stks (StackClassical cvs) )
        = showString "<<I- " . shows s .
          showString "(" . shows di . showString ") " .
          showString ": " .
          showList (zip cvs stks) . showString " ->>"
    showsPrec _  (QuantumStack s di stks (StackQubit qbs) )
        = showString "<<Q- " . shows s .
          showString "(" . shows di . showString ") " .
          showString ": " . showList qbs .
          showList (zip  qbs  stks) .
          showString    " ->>"
    showsPrec _  (QuantumStack s di stks (StackData cads))
        = showString  "<<C- " . shows s .
          showString "(" . shows di . showString ") " .
          showString "(" .
          showList (zip cads stks)  .
          showString ") ->>"


instance (Num b) =>  Eq (StackDescriptor b) where
    (==) StackZero StackZero = True
    (==) StackZero (StackValue b) = b == fromInteger 0
    (==) (StackValue b) StackZero = b == fromInteger 0
    (==) (StackValue b) (StackValue b') = b == b'
    (==) (StackClassical cvs) (StackClassical cvs')
        = foldl' (&&) True $ zipWith (==) cvs cvs'
    (==) (StackQubit qbs1) (StackQubit qbs2)
        = foldl' (&&) True $ zipWith (==) qbs1 qbs2
    (==) (StackData cads)  (StackData cads')
        = foldl' (&&) True $ zipWith (==) cads cads'
    (==) _ _ = False


instance (Num b, Eq b) =>  Eq (QuantumStack b) where
    (==) qs1 qs2
        = (descriptor qs1 == descriptor qs2) &&
          (address qs1 == address qs2) &&
          foldl' (&&) True (zipWith (==) (subStacks qs1) (subStacks qs2))



\end{code}
\end{singlespace}
```

```
\caption{Haskell definition of the quantum stack}\label{fig:haskellDefinitionOfQstack}
\end{figure}}


\emph{Quantum} data consists of \qubit{}s. These
are stored in the
quantum stack isomorphically to their density matrix notation.
Assuming a basis of \ket{0} an../QSM/QuantumStack/QSTransforms.lhs:215:38:
d \ket{1}, either three or
four entries are required
to store a \qubit. This is the same as the
density matrix for the
\qubit{} $\alpha\ket{0} + \beta\ket{1}$ which is
{\begin{singlespace}
$\begin{pmatrix}
\alpha\bar{\alpha}& \alpha\bar{\beta}\\
\beta\bar{\alpha}& \beta\bar{\beta}
\end{pmatrix}$\end{singlespace}}.

\emph{Probabilistic} data in the stack may be an \Int,
\Bool{} or \Datatype.
These are represented as multi-branched trees,
with one branch per
integer or logical value or data constructor.


The leaves of the tree are single valued numeric
values. These values
represent the actual value of the density matrix.

Finally, when all the leaves below a branch are $0$, a
special representation is used to effect sparseness.

Formerly, this  code was written assuming a generic |Basis| which
may have more than two elements. For example if
a |Basis| had three
elements,  a
\qubit{} representation would require six values.
(A $3\times 3$ matrix
with the below diagonal elements not stored).

Now, to achieve the best in speed that we can in the emulator, I
have migrated to a presumed basis of \ket{0} and \ket{1}. This
significantly reduces the complexity of the representation and
of the processing associated with the quantum stack.

Previously, the quantum stack was parametrized by both the basis and
the complex number type used for \qubit coefficients. Now, we
parametrize only by the coefficient type.


\begin{singlespace}
\begin{code}



qApply :: (QuantumStack b->QuantumStack b) ->
          QuantumStack b -> QuantumStack b
qApply g (QuantumStack nm od sstaks d)
       = QuantumStack nm od (map g sstaks) d
```

```
qAccessor :: (Quantum b) => Basis -> Basis ->
            QuantumStack b -> QuantumStack b
qAccessor Z Z qs@(QuantumStack _ _ ss (StackQubit qbs))
    | (Z,Z) `elem` qbs    =  head ss
    | otherwise           = zerostack
qAccessor Z O qs@(QuantumStack _ _ ss (StackQubit qbs))
    | (Z,O) `elem` qbs    = snd $ head $ filter (\q -> (Z,O) == fst q) $ zip qbs ss
    | otherwise           = zerostack
qAccessor O Z qs@(QuantumStack _ diag ss (StackQubit qbs))
    | diag                = (cjgtTranspose . qAccessor Z O) qs
    | (O,Z) `elem` qbs    = snd $ head $ filter (\q -> (O,Z) == fst q) $ zip qbs ss
    | otherwise           = zerostack
qAccessor O O  qs@(QuantumStack _ diag ss (StackQubit  qbs))
    | (O,O) `elem` qbs    = snd $ head $ filter (\q -> (O,O) == fst q) $ zip qbs ss
    | otherwise           = zerostack


qAccessor _ _ _ = error "Can not use accessor on non-qubit stack."


zz :: (Quantum b)=> QuantumStack b -> QuantumStack b
zz = qAccessor Z Z

zo :: (Quantum b)=> QuantumStack b -> QuantumStack b
zo = qAccessor Z O

oz :: (Quantum b)=> QuantumStack b -> QuantumStack b
oz = qAccessor O Z

oo :: (Quantum b)=> QuantumStack b -> QuantumStack b
oo = qAccessor O O
--qElems :: (Quantum b) => QV b -> [QuantumStack b]
--qElems (QV True q1 q2 _ q4) = [q1, q2, cjgt q2, q4]
--qElems (QV False q1 q2 q3 q4) = [q1, q2, q3, q4]


\end{code}
\end{singlespace}

The functions |val| and |valMaybe| are
accessor functions that will return a
quantum substack given a substack of a |QV| element.
%if false
\begin{code}
val :: (Quantum b)=>(Basis,Basis) ->
       QuantumStack b  ->
       QuantumStack b
val (a,b) = qAccessor a b



setAddress  ::  StackAddress -> QuantumStack b -> QuantumStack b
setAddress  s qs = qs{address=s}


secondAddress :: QuantumStack b -> StackAddress
secondAddress (QuantumStack _ _ sstacks _)
           = foldToAddress $ map address sstacks
```

```
\end{code}
%endif


The accessor function |qvalues| retrieves the |QV| sub element of
a |QuantumStack|.
%if codeOnly || showSupportFunctionDetail
\begin{code}



qvalues :: (Quantum b)=> QuantumStack  b ->
          [QuantumStack b]
qvalues  qs@(QuantumStack _ _ _ (StackQubit _))
    = [zz qs, zo qs, oz qs, oo qs]
qvalues  qs@(QuantumStack _ _ _ StackZero) = replicate 4 zerostack


qvaluesDiag :: (Quantum b)=> QuantumStack  b ->
          [QuantumStack b]
qvaluesDiag  qs@(QuantumStack _ True _ (StackQubit _))
    = [zz qs, zo qs,  oo qs]
qvaluesDiag  qs@(QuantumStack _ True _ StackZero) = replicate 3 zerostack
qvaluesDiag qs  = qvalues qs


diagq :: QuantumStack b -> QuantumStack b
diagq q = q{onDiagonal = True}

undiagq :: QuantumStack b -> QuantumStack b
undiagq q = q{onDiagonal = False}

setQvalues :: (Quantum b)=> [QuantumStack b] ->
              QuantumStack b -> QuantumStack b
setQvalues vals@[zzv,zov,  oov] qs
           | onDiagonal qs
               = let (ss, qbvals) = dropzsQ [diagq zzv, undiagq zov,diagq oov] diagQbs
                 in qs{subStacks = ss,  descriptor = qbvals}
           | otherwise
               = let vals' = [zzv, zov, cjgtTranspose zov, oov]
                     (ss, qbvals) = dropzsQ (map undiagq vals') fullQbs
                 in qs{subStacks = ss,  descriptor = qbvals}
setQvalues vals@[zzv,zov, ozv, oov] qs
           | onDiagonal qs
               = let (ss, qbvals) = dropzsQ [diagq zzv, undiagq zov,diagq oov] diagQbs
                 in qs{subStacks = ss,  descriptor = qbvals}
           | otherwise
               = let (ss, qbvals) = dropzsQ (map undiagq vals) fullQbs
                 in qs{subStacks = ss,  descriptor = qbvals}

setQvalues val qs
    | length val == descLength (descriptor qs) = qs{subStacks = val}
    | otherwise = error $ "Setting qv substack incorrectly - vals = "++
                          (showList val ", qs = "++show qs)


setSubStacks ::  (Quantum b)=> [QuantumStack b] ->
                QuantumStack b -> QuantumStack b
```

```
setSubStacks subs qs
   | isStackLeaf qs  = qs
   | isStackQubit qs = setQvalues subs qs
   | otherwise
     = case (descriptor qs) of
          StackClassical cvals ->
              let (ss, cvals') = dropzsC subs cvals
              in qs{subStacks = ss, descriptor = cvals'}
          StackData dvals ->
              let (ss, dvals') = dropzsD subs dvals
              in qs{subStacks = ss, descriptor = dvals'}


trimSubStacks  :: (Quantum b)=>QuantumStack b->
                  QuantumStack b
trimSubStacks  qs = setSubStacks (subStacks qs) qs


dropzsC :: (Num b) => [QuantumStack b] -> [ClassicalData] ->
          ([QuantumStack b], StackDescriptor b)
dropzsC qs bs =
    case dropzs' qs bs of
       []    -> ([], StackZero)
       lis   -> let (qs', bs') = unzip lis
                in (qs', StackClassical bs')


dropzsD :: (Num b) => [QuantumStack b] -> [(Constructor,[StackAddress])] ->
          ([QuantumStack b], StackDescriptor b)
dropzsD qs bs =
    case dropzs' qs bs of
       []    -> ([], StackZero)
       lis   -> let (qs', bs') = unzip lis
                in (qs', StackData bs')

dropzsQ ::  (Num b) => [QuantumStack b] -> [(Basis,Basis)] ->
          ([QuantumStack b], StackDescriptor b)
dropzsQ qs bs =
    case dropzs' qs bs of
       []    -> ([], StackZero)
       lis   -> let (qs', bs') = unzip lis
                in (qs', StackQubit bs')

dropzs' :: (Num b) => [QuantumStack b] -> [a] ->
          [(QuantumStack b, a)]
dropzs' [] _ = []
dropzs' (q:qs) (b:bs)
       | isStackZero q  = dropzs' qs bs
       | otherwise      = (q,b):dropzs' qs bs

\end{code}
%endif


%if codeOnly || showMinorFunctions
\begin{code}

trace  :: (Quantum b)=>
          QuantumStack  b ->
          b
trace  (QuantumStack _ _ _ StackZero)          =  fromInteger 0
```

```
trace  (QuantumStack _ _ _ (StackValue b))      =  b
trace  qs@(QuantumStack _ _ _ (StackQubit _))   =  trace (zz qs) + trace (oo qs)
trace  qs@(QuantumStack _ _ sstks _)            =  foldr (+) (fromInteger 0) $
                                                   map trace sstks


branchCount ::  QuantumStack b ->     Int
branchCount = length . subStacks



eraseZeroes :: QuantumStack b -> Bool
eraseZeroes (QuantumStack _ _ _ StackZero) = False
eraseZeroes qs = True




depth :: QuantumStack b -> Int
depth (QuantumStack _ _ _ StackZero) = 1
depth (QuantumStack _ _ _ (StackValue _)) = 1
depth qs = 1 + foldl' max 0 (map depth  (subStacks qs))


addresses ::  Int -> QuantumStack b->[StackAddress]
addresses 1 qb = [address qb]
addresses n qs
    | n > 0 =
        address qs : addresses (n-1) (subTreeWithDepth (n-1) $ subStacks qs)
    | otherwise  = error "Can not get further addresses from Zero or leaves"

allAddresses ::  QuantumStack b->[StackAddress]
allAddresses qs =
    address qs : concatMap allAddresses (subStacks qs)



subTreeWithDepth ::  Int ->
                     [QuantumStack b] -> QuantumStack b
subTreeWithDepth n qstks
    = qHead "subTreeWithDepth " deepStacks
      where deepStacks = filter (\a -> n <= depth a) qstks
\end{code}
%endif


The function |getQubits| filters its first argument (a list of
|StackAddress|es) by determining which of them are \qubit{}s in the
current quantum stack. The filtered list is then returned.

The filtering function |isQubit| returns true if the top node is
a \qubit{} with the correct address, otherwise it recurses down the quantum stack.

{\begin{singlespace}
\begin{code}
getQubits :: [StackAddress] -> QuantumStack b ->
           [StackAddress]
getQubits nms q = filter (flip isQubit q) nms
```

```
isTypeX :: StackDescriptor b -> StackAddress ->
           QuantumStack b -> Bool
isTypeX des nm qs
       = (nm == address qs && des =~= descriptor qs) ||
         foldl' (||) False (map (isTypeX des nm) $ subStacks qs)

isQubit ::StackAddress -> QuantumStack b -> Bool
isQubit = isTypeX (StackQubit [])


\end{code}
\end{singlespace}}

The following functions are similiar to the |isTypeX| based ones
except that they just operate on the top stack element.

{\begin{singlespace}
\begin{code}

isStackClassical :: QuantumStack b -> Bool
isStackClassical (QuantumStack _ _ _ (StackClassical  _))  = True
isStackClassical _               = False

isStackQubit ::  QuantumStack b -> Bool
isStackQubit (QuantumStack _ _ _ (StackQubit _) )  = True
isStackQubit _               = False


isStackData :: QuantumStack b -> Bool
isStackData (QuantumStack _ _ _ (StackData _))  = True
isStackData _               = False

isStackValue :: QuantumStack b -> Bool
isStackValue (QuantumStack _ _ _ (StackValue _))  = True
isStackValue _               = False

isStackZero :: (Num b) => QuantumStack b -> Bool
isStackZero qs = let desc = descriptor qs
                 in desc == StackZero || (desc == (StackValue $ fromInteger 0))

isStackLeaf ::  QuantumStack b -> Bool
isStackLeaf qs = case descriptor qs of
                     StackZero      -> True
                     StackValue _   -> True
                     _              -> False

isDiagQubit :: QuantumStack b -> Bool
isDiagQubit  qs = case   descriptor qs of
                         StackQubit ((Z,Z):_)      -> True
                         StackQubit ((O,O):_)      -> True
                         _                        -> False


\end{code}
\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}
\begin{code}


isDtype :: StackAddress -> QuantumStack b -> Bool
isDtype  = isTypeX (StackData [])

\end{code}
\end{singlespace}


\begin{code}

stackUnionWith :: Ord a =>
                  (QuantumStack b -> QuantumStack b -> QuantumStack b) ->
                 [(a,QuantumStack b)] -> [(a,QuantumStack b)] ->
                 [(a,QuantumStack b)]
stackUnionWith f [] [] = []
stackUnionWith f [] bs = bs
stackUnionWith f aas [] = aas
stackUnionWith f ((a,q1):aas) ((b,q2):bbs)
    | a == b = (a, f q1 q2): stackUnionWith f aas bbs
    | a < b = (a, q1): stackUnionWith f aas ((b,q2):bbs)
    | a > b = (b, q2): stackUnionWith f ((a,q1):aas) bbs

cjgt :: (Quantum  b) =>
        QuantumStack  b ->
        QuantumStack  b
cjgt qs@(QuantumStack _ _ _ StackZero)
    = qs
cjgt qs@(QuantumStack _ _ _ (StackValue b))
    = qs{descriptor = StackValue $ conjgt b}
cjgt qs       = qApply cjgt qs

cjgtTranspose :: (Quantum  b) =>
        QuantumStack  b ->
        QuantumStack  b
cjgtTranspose qs@(QuantumStack _ _ _ StackZero)
    = qs
cjgtTranspose qs@(QuantumStack _ _ _ (StackValue b))
    = qs{descriptor = StackValue $ conjgt b}
cjgtTranspose qs@(QuantumStack _ False _ (StackQubit _))
     = let [a,b,c,d] = map cjgtTranspose $ qvalues qs
       in setQvalues [a,c,b,d] qs
--cjgtTranspose qs@(QuantumStack _ True _ (StackQubit _))
--     = let [a,b,d] = map cjgtTranspose $ qvaluesDiag qs
--       in setQvalues [a,b,d] qs
cjgtTranspose qs       = qApply cjgtTranspose qs


zerostack :: QuantumStack b
zerostack = QuantumStack noAddress False [] StackZero

zerovaluedescriptor :: (Num b) =>StackDescriptor b
zerovaluedescriptor  = StackValue (fromInteger 0)
```

```
{- unused
equal :: (Eq a) => [a] -> Bool
equal [] = True
equal (a:aas) = foldl' (&&) True $ List.map ((==) a)  aas
-}
\end{code}
\end{singlespace}
}
```