Programming with classical quantum datatypes

Robin Cockett & Brett Giles

(robin,gilesb)@cpsc.ucalgary.ca

University of Calgary

Yet another Quantum Programming Language (QPL)...

- Linear type system with (classically controlled) datatypes
- No higher-order constructs ...
- Only built-in types are **Qbit** and **Int**.
- Built-in unitary transformations.
- A QPL program consists of function definitions and “main” instructions.
- Compiles to Quantum Stack Machine (QSM).
- Quantum assembler calculates density matrix.

Semantics ...

Countably infinite bi-products of completely positive matrices with trace less than or equal to 1.

$$\mathsf{fHilb} \to CPM(\mathsf{fHilb}) \to \mathsf{Mat}_\infty(CPM(\mathsf{fHilb}))$$

Similar to the semantics of Peter Selinger's QPL.

This QPL ...

- Can declare (classically controlled) datatypes (e.g . lists of Quantum bits).
- Linear variables are *unscoped*: live from introduction to first use (cannot be used twice).
- Classical data extracted from linear variables by **use** command.
- Classical variables are *scoped*: live to end of a block (multiple uses permitted).
- Quantum bits can be measured, Integers have usual arithmetic functions ....

Example datatype declarations

```
type  Bool = {True | False}

type List a = {Nil | Cons (a, (List a))}

type Either a b = {Left (a) | Right (b)}

type Maybe a = {Nothing | Just (a)}

type BinTree a = {Leaf (a) |
                  Br ((BinTree a), (BinTree a))}
```

Language constructs continued ...

Function declaration

$$\begin{aligned} \textbf{rotate} :: (n : \textbf{Int}, x_1 : \textbf{Qbit}, x_2 : \textbf{List(Qbit)}; \\ y_1 : \textbf{Qbit}, y_2 : \textbf{List(Qbit)}) \\ = \quad \{\textit{statements}\} \end{aligned}$$

Function calls:

- $\textbf{rotate}(n, x, y; x, y);$
- $(x, y) = \textbf{rotate}(n, x, y);$
- $\textbf{rotate}(n)\ x\ y;$
- $\textbf{Had}\ q; \textbf{Not}\ q; \textbf{Not}\ q_1 <= q_2; \ldots$

Example Program: coin flip

```
#Import Prelude.qpl

cflip ::(; b:Bool) =
{  q = |0>;
   Had q;
   measure q of
     |0> => {b = False}
     |1> => {b = True};
}


main::() =
{  b = cflip(); }
```

Language construct: measurement

$$\textbf{measure } q \textbf{ of } \begin{array}{lcl} |0\rangle & \Rightarrow & \{\ldots\} \\ |1\rangle & \Rightarrow & \{\ldots\}; \end{array}$$

. . . measurement is a control construct . . .

Example Program: Teleport

```
prepare :: (;a:Qbit,b:Qbit)
= { a = |0>; b = |0>;
    Had a;
    Not b <= a } //Controlled Not

teleport :: (x:Qbit; b:Qbit)
= { (a,b) = prepare();
    Not a <= x;
    Had x;
    measure a of |0> => {}
                 |1> => {Not b};
    measure x of |0> => {}
                 |1> => {RhoZ b}  }
```

Language constructs continued ...

Loops: ... handled by recursion.

Pattern matching:

$$\textbf{case } q \textbf{ of} \quad \begin{array}{lcl} \textbf{nil} & \Rightarrow & \{\ldots\} \\ \textbf{cons}(x, xs) & \Rightarrow & \{\ldots\}; \end{array}$$

Example Program: append

```
#Import Prelude.qpl

append :: (x:List(a), y:List(a) ; z:List(a)) =
{  case x of
       Nil          => { z = y }
       Cons (v,vs)  => { z = Cons(v,append(vs,y)) }
}
```

Language constructs continued ...

Discarding:

$$\textbf{discard}\ q_1, \ldots, q_n;$$

implicit discarding of variables on joining of control paths (with warning).

Linear assignment is basic:

$$q = \textit{expression};$$

q is a linear variable ...
There is syntactic sugar for classical assignment (see below).

Using classical data ...

$$\textbf{use}\ x_1, \ldots, x_n\ \textbf{in}\ \{\textit{statements}\}$$

Syntactic sugar:

$$x := \textit{expr}; \ldots; \quad \equiv \quad x = \textit{expr};\ \textbf{use}\ x\ \textbf{in}\ \{\ldots;\}$$

$$\textbf{use}\ x_1, \ldots, x_n; \ldots; \quad \equiv \quad \textbf{use}\ x_1, \ldots, x_n\ \textbf{in}\ \{\ldots;\}$$

Example Program: rotate

```
rotate ::(n:Int, h:Qbit, z:List (Qbit);
                 h:Qbit, z:List (Qbit))
= { case z of
        Nil =>
          { discard n;
            z = Nil }
        Cons(x,y) =>
          { use n in
             { R(n) x h;
               m = n+1 };
            rotate(m) h y;
            z = Cons(x,y) }
}
```

Example Program: Quantum Fourier Transform

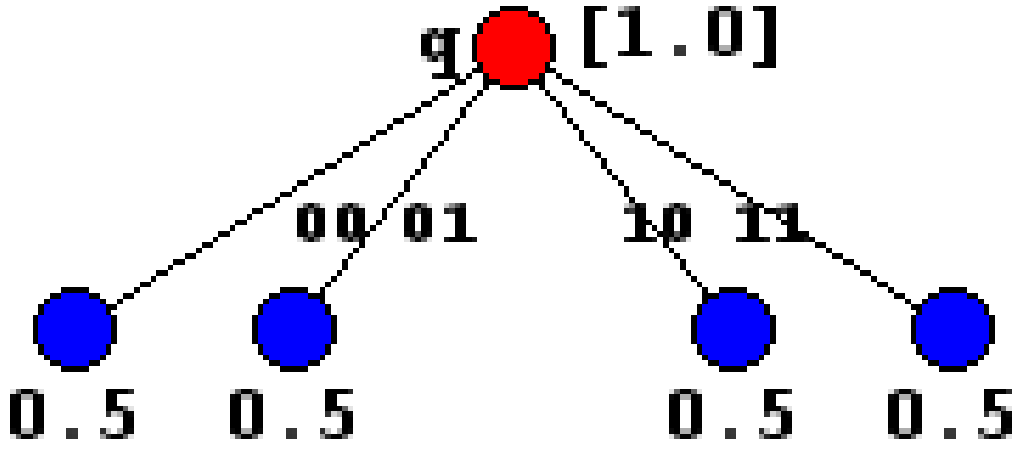
```
qft :: (qf:List(Qbit); qf:List(Qbit)) =
{
   case qf of
     Nil => {qf = Nil}
     Cons (head,tail) => {
        { Had head;
          rotate(2) head tail;
          qft tail;
          qf = Cons(head,tail) }
}
```

A Quantum stack machine

- The machine state consists of the quantum stack, a classical stack, and a dump.
- The quantum stack is represented as a tree, with three types of nodes (Qbit, Datatype, Int) and values at the leaves (zero probability branches are not represented).
- Operations are done primarily on the top node or top few nodes. Rotation of the tree allows us to bring a node to the top.
- The dump is used to hold intermediate data during instruction execution.

Qbit Nodes

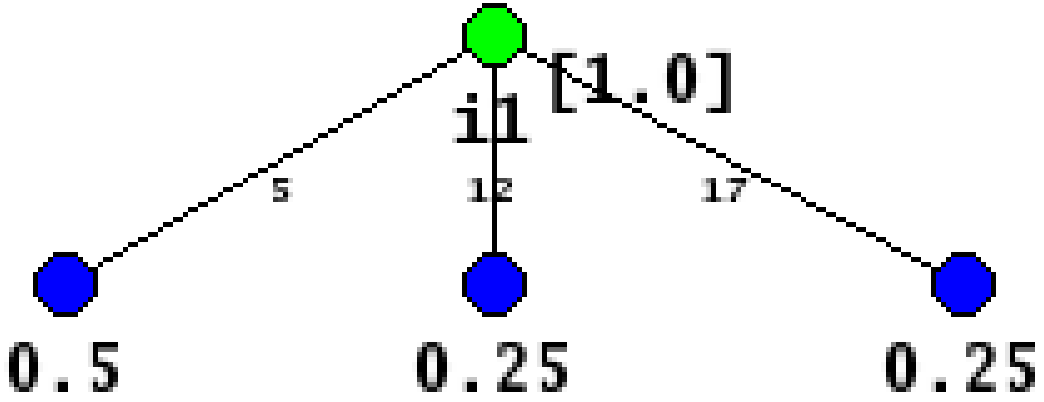
A **Qbit** is represented by a node with four branches, labeled by 00, 01, 10 and 11 (equivalent to the density matrix notation of a **Qbit**).



Note that physically, we only store those branches with non-zero values at the leaves.

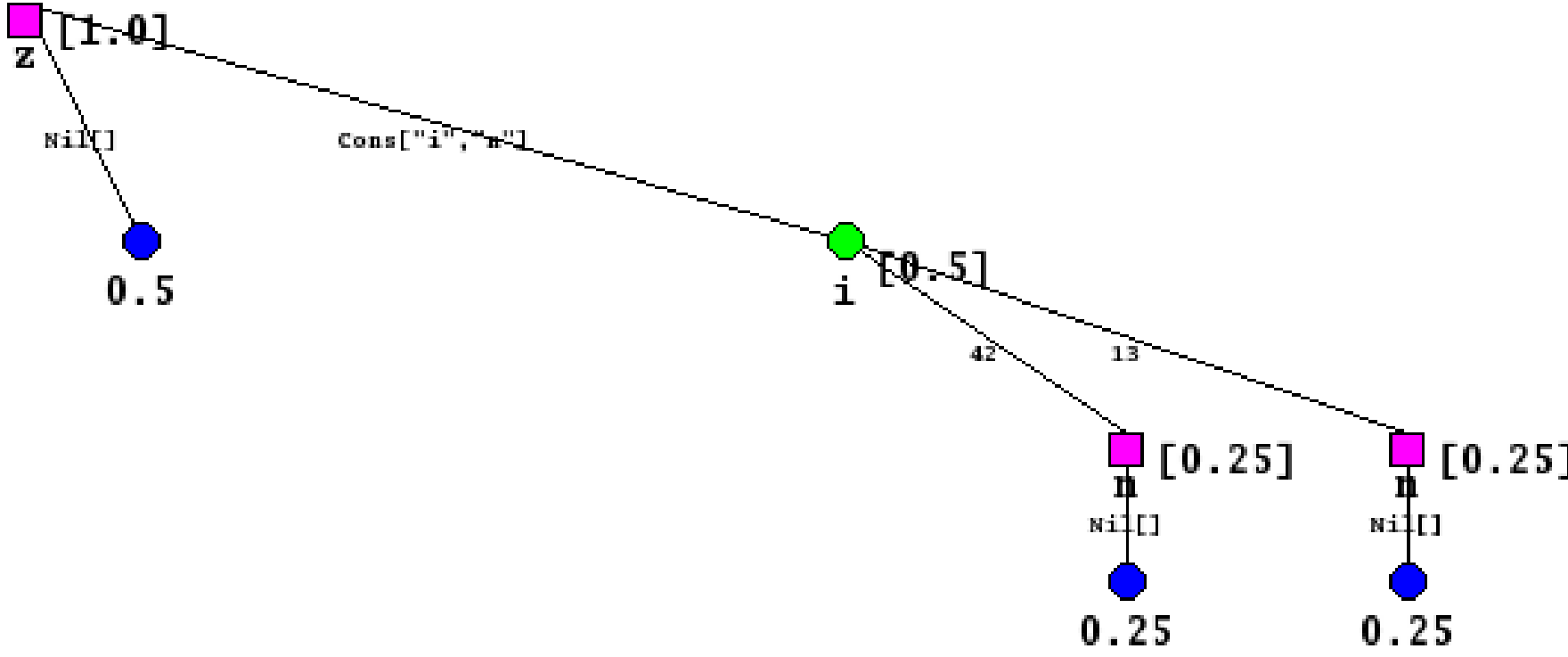
Int nodes

Classical data is represented by a node with an arbitrary (finite) number of branches labeled by classical values (e.g. **Int** values):



Datatype nodes

A node with a branch for each constructor which is labeled by the constructor name and the variables the construction binds.



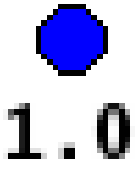
Binding ...

Variables bound along a branch of a node *cannot* be rotated above their point of binding ... so can not be manipulated directly in the Quantum stack.

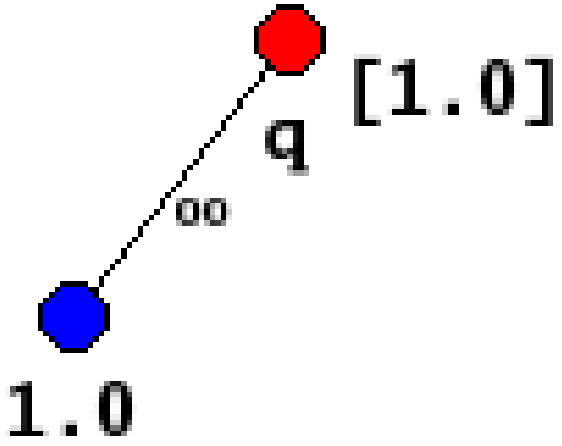
Example program evaluation: coin flip ...

```
#Import Prelude.qpl

cflip ::(;q:Qbit,r:Bool) =
{  q = |0>;
   Had q;
   measure q of
     |0> => {q = |0>;r = True}
     |1> => {q = |1>;r = False};
}


main::() =
{  (q,b) = cflip(); }
```

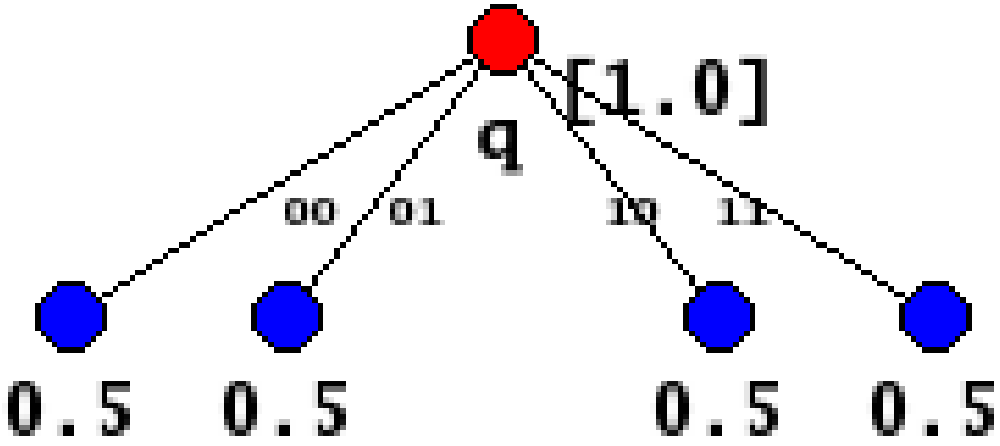
Empty stack



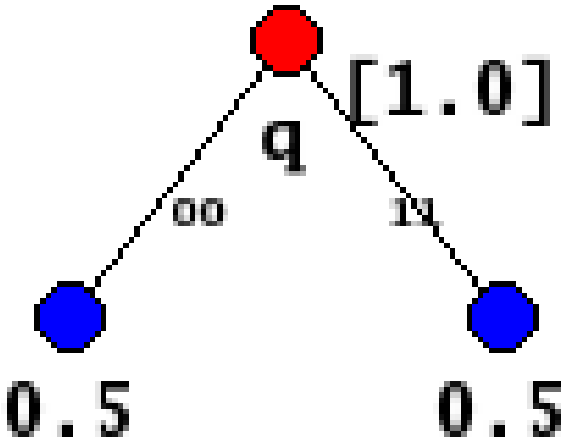
Introduce Qbit 0



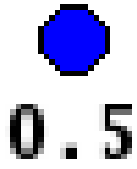
Hadamard



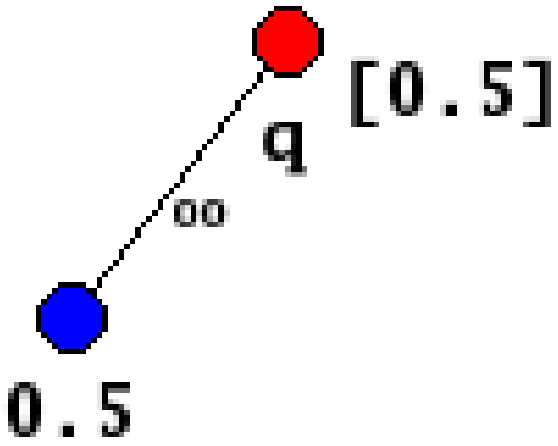
Measure



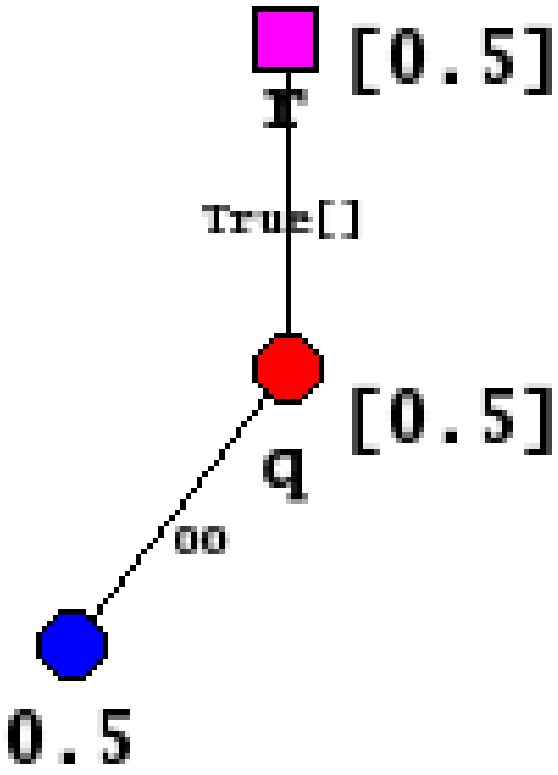
Left branch



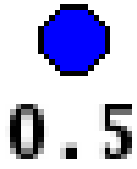
Left branch: introduce Qbit 0



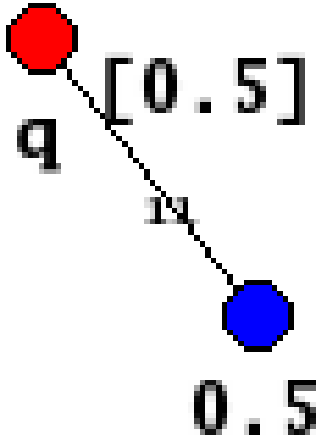
Left branch: introduce True



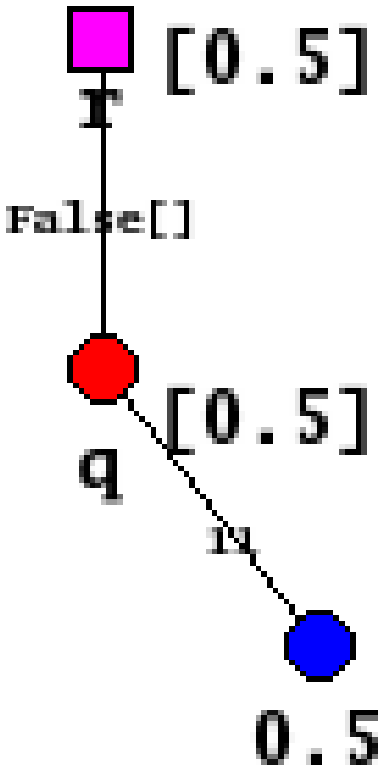
Swap to right branch



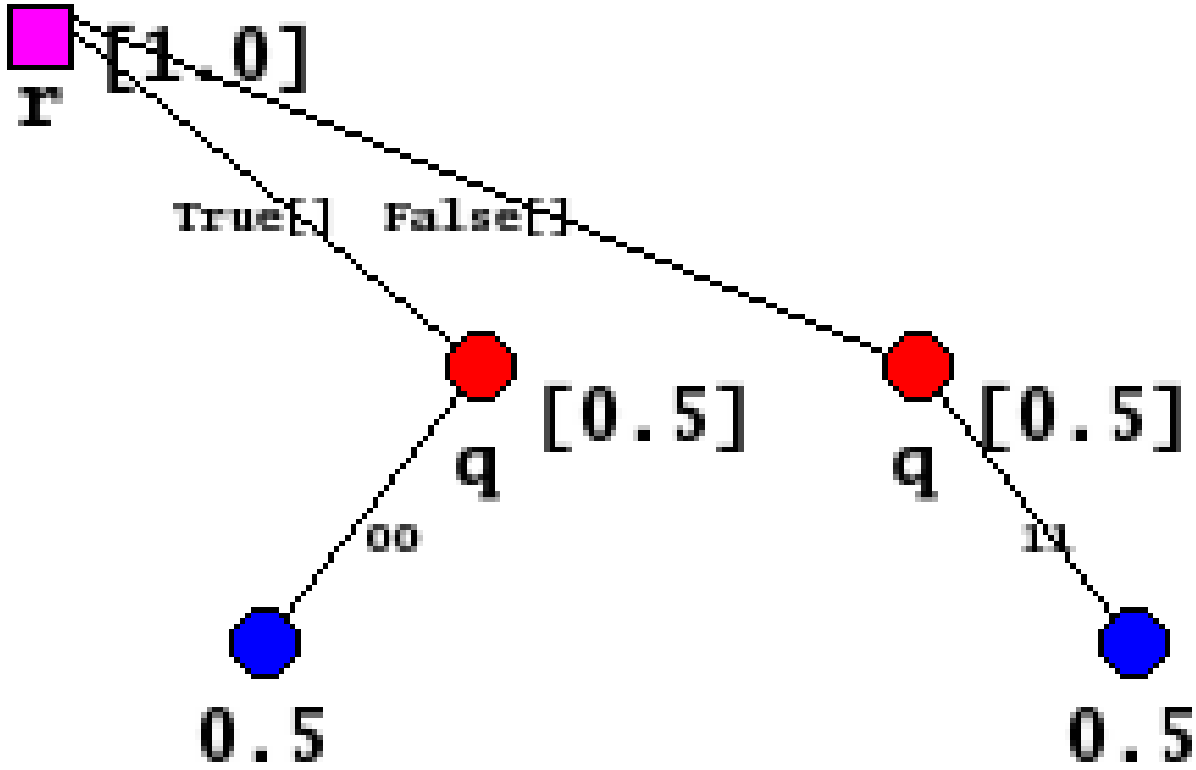
Right branch: introduce Qbit 1



Right branch: introduce False



Merge



Quantum Stack Machine Instructions

QPL programs compile to Quantum Stack Machine (QSM) code.

$$\text{QPL} \longrightarrow \text{QSM}$$

QSM code is run by the QSM-assembler.
The design of an efficient yet basic instruction set for this machine has been an issue ...

QSM Instructions: node construction

`QLoad` x $|k\rangle$: creates a node called x with a single branch labeled $|k\rangle$ on top of the Quantum stack.

`QCons` x `cons`: creates a node called x with a single branch labeled `cons` (the specific constructor) on top of the Quantum stack.

`QMove` x: removes the top value of the classical stack and creates, on top of the Quantum stack, a node called x with a single branch labeled by that classical value.

`QBind` z: expects a node with a single branch and binds the variable z down that branch.

Construction Transitions

$$(\text{QLoad } x\ |k\rangle{:}\mathcal{C}, S, Q, D) \implies (\mathcal{C}, S, x{:}[|k\rangle \to Q], D)$$

$$(\text{QCons } x\ c{:}\mathcal{C}, S, Q, D) \implies (\mathcal{C}, S, x{:}[c\{\} \to Q], D)$$

$$(\text{QBind } z_0{:}\mathcal{C}, S, x{:}[c\{z'_1, \ldots, z'_n\} \to Q], D) \implies (\mathcal{C}, S, x{:}[c\{z'_0, z'_1, \ldots, z'_n\} \to Q[z'_0/z_0]], D)$$

$$(\text{QMove } x{:}\mathcal{C}, v{:}S, Q, D) \implies (\mathcal{C}, S, x{:}[\overline{v} \to Q], D)$$

QSM Instructions: node destruction

`QUnbind` x: Expects the stack have a node with a single branch on top with a list of bound variables. The first bound variable is removed and renamed to x to make it free.

`QDiscard`: Discards the top node by merging its "classical" branches.

Destruction Transitions

$$(\text{QDiscard}{:}\mathcal{C}, S, x{:}[|k\rangle \to Q], D) \implies (\mathcal{C}, S, Q, D)$$

$$(\text{QDiscard}{:}\mathcal{C}, S, x{:}[c\{\} \to Q], D) \implies (\mathcal{C}, S, Q, D)$$

$$(\text{QDiscard}{:}\mathcal{C}, S, x{:}[\bar{v} \to Q], D) \implies (\mathcal{C}, v{:}S, Q, D)$$

$$(\text{QUnbind } y{:}\mathcal{C}, S, x{:}[c\{z'_1, \ldots, z'_n\} \to Q], D) \implies (\mathcal{C}, S, x{:}[c\{z'_2, \ldots, z'_n\} \to Q[y/z'_1]], D)$$

$(|k\rangle \in \{|0\rangle, |1\rangle\})$

QSM Instructions: stack manipulation

`QPullup` x: Rotates the node labeled x to the top of the Quantum stack.

`QApply` 0 $\mathbf{Had}$: Expects 1 **Qbit** on top of the Quantum stack and applies the Hadamard transformation to it.

`QApply` 1 $\mathbf{Rotate}$: Parametrizes the $\mathbf{Rotate}$ transform by the top element of the classical stack and then applies it to the 2 **Qbits** on the top of the stack.

`QName` x y: Renames the (visible) node x to y.

Manipulation Transitions

$$(\text{QPullup } x{:}\mathcal{C}, S, Q, D) \implies (\mathcal{C}, S, \mathsf{pull}(x, Q), D)$$

$$(\text{QApply } n\ t{:}\mathcal{C}, v_1{:}\cdots{:}v_n{:}S, Q, D) \implies (\mathcal{C}, v_1{:}\cdots{:}v_n{:}S, \mathsf{transform}([v_1, \ldots, v_n], t, Q), D)$$

$$(\text{QName } x\ y{:}\mathcal{C}, S, Q, D) \implies (\mathcal{C}, S, Q[y/x], D)$$

QSM Instructions: Quantum control; Use; Split; Measure

`Use label`: Expects top of the Quantum stack to be classical. For each branch put this value on the classical stack, perform the computation at `label`, which is ended by `EndQC`, and merge the results.

`Split` $[\texttt{Nil} \rightarrow \texttt{label}_1, \texttt{Cons} \rightarrow \texttt{label}_2]$: Expects a datatype node on top of the Quantum stack. Executes the code at $\texttt{label}_i$, which is ended with `EndQC`, on each branch with a matching label and merges the results.

`Meas` $\texttt{label}_1$ $\texttt{label}_2$: Expects a **Qbit** node on top of the Quantum stack. Executes the code at $\texttt{label}_1$, which is ended with `EndQC`, on the $|0\rangle$ branch and $\texttt{label}_2$ on the $|1\rangle$ branch and merges the results.

Transitions for Quantum Control - Use

$$(\text{Use } \triangleright\mathcal{C}_U{:}\mathcal{C}, S, x{:}[\bar{v}_i \to Q_i], D) \implies (\text{EndQC}, S, 0, \mathtt{Qc}(S, [(x_i{:}v_i \to Q_i, \triangleright\mathcal{C}_U)], \triangleright\mathcal{C}, 0){:}D)$$

$$(\text{EndQC}, S', Q, \mathtt{Qc}(S, [(x_i{:}v_i \to Q_i, \triangleright\mathcal{C}_U)]_{i=j,\ldots,m}, \triangleright\mathcal{C}, Q'){:}D) \implies (\mathcal{C}_U, S, x_j, \mathtt{Qc}(S, [(x_i{:}v_i \to Q_i, \triangleright\mathcal{C}_U)]_{i=j+1,\ldots,m}, \triangleright\mathcal{C}, Q + Q'){:}D)$$

$$(\text{EndQC}, S', Q, \mathtt{Qc}(S, [], \triangleright\mathcal{C}, Q'){:}D) \implies (\mathcal{C}, S, Q + Q', D)$$

Transitions for Quantum Control - Split

$$(\text{Split } [(c_i, \triangleright\mathcal{C}_i)]{:}\mathcal{C}, S, x{:}[c_i\{V_i\} \to Q_i], D) \implies (\text{EndQC}, S, 0, \texttt{Qc}(S, [(x_i{:}c_i\{V_i\} \to Q_i, \triangleright\mathcal{C}_i)], \triangleright\mathcal{C}, 0){:}D)$$

$$(\text{EndQC}, S', Q, \texttt{Qc}(S, [(x_i{:}c_i\{V_i\} \to Q_i, \triangleright\mathcal{C}_i)]_{i=j,\ldots,m}, \triangleright\mathcal{C}, Q'){:}D) \implies (\mathcal{C}_j, S, x_j, \texttt{Qc}(S, [(x_i{:}c_i\{V_i\} \to Q_i, \triangleright\mathcal{C}_i)]_{i=j+1,\ldots,m}, \triangleright\mathcal{C}, Q + Q'){:}D)$$

$$(\text{EndQC}, S', Q, \texttt{Qc}(S, [], \triangleright\mathcal{C}, Q'){:}D) \implies (\mathcal{C}, S, Q + Q', D)$$

Transitions for Quantum Control - Measure

$$(\text{Meas} \triangleright\mathcal{C}_0 \triangleright\mathcal{C}_1{:}\mathcal{C}, S, x{:}[|0\rangle \to Q_0, |1\rangle \to Q_1, \langle\rangle \to Q], D) \implies (\text{EndQC}, S, 0, \mathtt{Qc}(S, [(x_k{:}|k\rangle \to Q_k, \triangleright\mathcal{C}_k)]_{k\in\{0,1\}}, \triangleright\mathcal{C}, 0){:}D)$$

$$(\text{EndQC}, S', Q, \mathtt{Qc}(S, [(x_k{:}|k\rangle \to Q_k, \triangleright\mathcal{C}_k)]_{k\in\{0,1\}}, \triangleright\mathcal{C}, Q'){:}D) \implies (\mathcal{C}_0, S, x_0, \mathtt{Qc}(S, [(x_1{:}|1\rangle \to Q_1, \triangleright\mathcal{C}_1)], \triangleright\mathcal{C}, Q + Q'){:}D)$$

$$(\text{EndQC}, S', Q, \mathtt{Qc}(S, [(x_1{:}|1\rangle \to Q_1, \triangleright\mathcal{C}_1)], \triangleright\mathcal{C}, Q'){:}D) \implies (\mathcal{C}_1, S, x_1, \mathtt{Qc}(S, [], \triangleright\mathcal{C}, Q + Q'){:}D)$$

$$(\text{EndQC}, S', Q, \mathtt{Qc}(S, [], \triangleright\mathcal{C}, Q'){:}D) \implies (\mathcal{C}, S, Q + Q', D)$$

QSM Instructions: classical control

`Jump label`: Jumps to the code at `label`.

`CJump label`: Pops the top element of the classical stack if it is “true” jumps to the code at `label`.

`Call label`: Pushes the return code pointer onto the dump and jumps to the subroutine code.

`Return`: Pops the code label off the dump and continues execution of the code to which it points.

Transitions for Classical Control

$$(\text{Jump} \triangleright \mathcal{C}_J{:}\mathcal{C}, S, Q, D) \implies (\mathcal{C}_J, S, Q, D)$$

$$(\text{CJump} \triangleright \mathcal{C}_J{:}\mathcal{C}, \texttt{False}{:}S, Q, D) \implies (\mathcal{C}_J, S, Q, D)$$

$$(\text{CJump} \triangleright \mathcal{C}_J{:}\mathcal{C}, \texttt{True}{:}S, Q, D) \implies (\mathcal{C}, S, Q, D)$$

$$(\text{Call } n \triangleright \mathcal{C}_C{:}\mathcal{C}, v_1{:}\cdots{:}v_n{:}S, Q, D) \implies (\mathcal{C}_C, [v_1, \ldots, v_n], Q, R(S, \triangleright\mathcal{C}){:}D)$$

$$(\text{Return } n, v_1{:}\cdots{:}v_n{:}S', Q, R(S, \triangleright\mathcal{C}){:}D) \implies (\mathcal{C}, [v_1, \ldots, v_n]{:}S, Q, D)$$

QSM Instructions: classical

`CGet` n: Puts the n^{th} Classical stack value on top of the Classical stack.

`CApply` `ADD`: Pops the top values off the classical stack, applies the operation and pushes the result back onto the stack.

`CLoad` n: Pushes the constant value n onto the Classical stack.

`CPop`: Pops the top element off the Classical stack.

Transitions for Classical Operations

$$(\text{CPop}{:}\mathcal{C}, v{:}S, Q, D) \implies (\mathcal{C}, S, Q, D)$$

$$(\text{CGet } n{:}\mathcal{C}, v_1{:}\cdots{:}v_n{:}S, Q, D) \implies (c, v_n{:}v_1{:}\cdots{:}v_n{:}S, Q, D)$$

$$(\text{CApply } \mathsf{op}_n{:}\mathcal{C}, v_1{:}\cdots{:}v_n{:}S, Q, D) \implies (\mathcal{C}, \mathsf{op}_n(v_1, \ldots, v_n){:}S, Q, D)$$

$$(\text{CLoad } n\ {:}\mathcal{C}, S, Q, D) \implies (\mathcal{C}, n{:}S, Q, D)$$

Recursion

The machine stack is actually a stream of stacks, where stacks further down the stream correspond to progressive unfoldings of the recursion. The limit of the stream is the computation.
Instructions are lifted by the standard Kliesli lifting to produce a function between streams of Quantum stacks.

Conclusion

Where are we...

- State of QPL compiler: Working, generating code!
- State of quantum stack machine: Working with this instruction set, revising for controlled transforms.
- State of QPL programs: order finding for Shor's factoring algorithm compiling, and quantum search not done.

Where are we going...

- Expect to complete implementation this fall.
- Plan to make web runnable version available.
- Executable compiler and QPL programs will be down-loadable.