Types for Quantum Computing

Peter Selinger

Dalhousie University
Halifax, Canada

Why Quantum Programming Languages?

- For certain problems, quantum algorithms have an exponential speedup over best known classical algorithms.
- Most research in quantum computing has focused on algorithms and complexity theory.
- Quantum algorithms are traditionally described in terms of hardware: quantum circuits or quantum Turing machines.
- Want compositionality. Also, how do quantum features interact with other language features such as structured data, recursion, i/o, higher-order.

Part I: Quantum Computation

Linear Algebra Review

- Scalars $\lambda \in \mathbb{C}$, column vectors $u \in \mathbb{C}^n$, matrices $A \in \mathbb{C}^{n \times m}$.

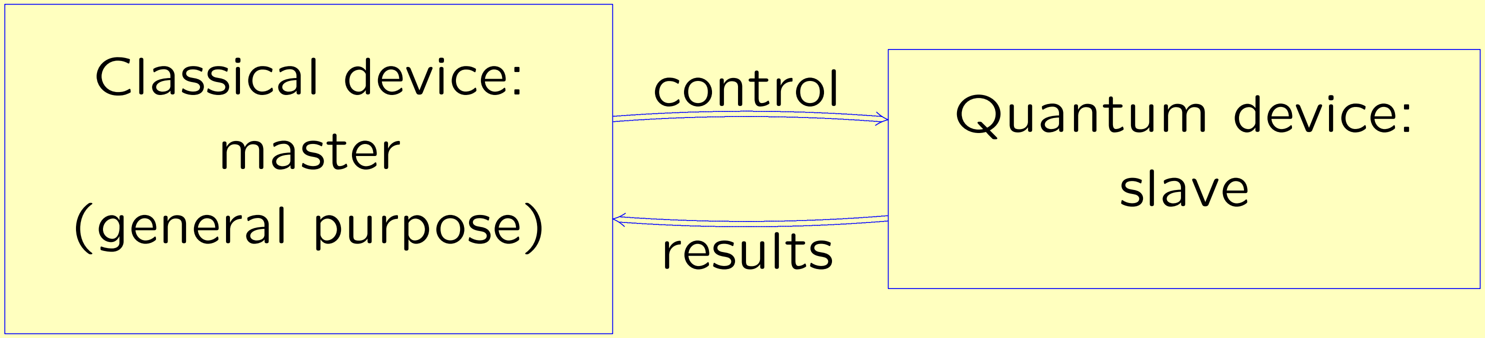
- Adjoint $A^* = (\overline{a_{ji}})_{ij}$, trace $\operatorname{tr} A = \sum_i a_{ii}$, norm $\|A\|^2 = \sum_{ij} |a_{ij}|^2$.

- Unitary matrix $S \in \mathbb{C}^{n \times n}$ if $S^* S = I$.
  Change of basis: $B = SAS^* \Rightarrow \operatorname{tr} B = \operatorname{tr} A$, $\|B\| = \|A\|$.

- Hermitian matrix $A \in \mathbb{C}^{n \times n}$: if $A = A^*$.
  Hermitian positive: $u^* A u \geq 0$ for all $u \in \mathbb{C}^n$.
  Diagonalization: $A = SDS^*$, S unitary, D real diagonal.

- Tensor product $A \otimes B$, e.g. $\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \otimes B = \begin{pmatrix} 0 & B \\ -B & 0 \end{pmatrix}$.

The QRAM abstract machine [Knill96]



- General-purpose classical computer controls a special quantum hardware device

- Quantum device provides a bank of individually addressable qubits.

- Left-to-right: instructions.

- Right-to-left: results.

Quantum computation: States

- state of one qubit: $\alpha|\mathbf{0}\rangle + \beta|\mathbf{1}\rangle$ (*superposition* of $|\mathbf{0}\rangle$ and $|\mathbf{1}\rangle$).

- state of two qubits: $\alpha|\mathbf{00}\rangle + \beta|\mathbf{01}\rangle + \gamma|\mathbf{10}\rangle + \delta|\mathbf{11}\rangle$.

- *independent*: $(a|\mathbf{0}\rangle + b|\mathbf{1}\rangle) \quad \otimes \quad (c|\mathbf{0}\rangle + d|\mathbf{1}\rangle) \quad =$
  $ac|\mathbf{00}\rangle + ad|\mathbf{01}\rangle + bc|\mathbf{10}\rangle + bd|\mathbf{11}\rangle$.

- otherwise *entangled*.

Lexicographic convention

Identify the basis states $|\mathbf{00}\rangle$, $|\mathbf{01}\rangle$, $|\mathbf{10}\rangle$, $|\mathbf{11}\rangle$ with the standard basis vectors

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix},$$

in the *lexicographic* order.

Note: we use *column vectors* for states.

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \delta \end{pmatrix} = \alpha|\mathbf{00}\rangle + \beta|\mathbf{01}\rangle + \gamma|\mathbf{10}\rangle + \delta|\mathbf{11}\rangle.$$

Quantum computation: Operations

- unitary transformation
- measurement

Some standard unitary gates

Unary:

$$N = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

$$V = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix},$$

$$W = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{i} \end{pmatrix},$$

Binary:

$$N_c = \left(\begin{array}{c|c} I & 0 \\ \hline 0 & N \end{array}\right),$$

$$H_c = \left(\begin{array}{c|c} I & 0 \\ \hline 0 & H \end{array}\right),$$

$$V_c = \left(\begin{array}{c|c} I & 0 \\ \hline 0 & V \end{array}\right),$$

$$W_c = \left(\begin{array}{c|c} I & 0 \\ \hline 0 & W \end{array}\right), \quad X = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Measurement

$$
\begin{tikzcd}
& \alpha|\mathbf{0}\rangle + \beta|\mathbf{1}\rangle \arrow[dl, "|\alpha|^2"', "\mathbf{0}"] \arrow[dr, "\mathbf{1}"', "|\beta|^2"] & \\
\alpha|\mathbf{0}\rangle & & \beta|\mathbf{1}\rangle
\end{tikzcd}
$$

Two Measurements

$$
\begin{tikzcd}[column sep=small]
 & & & \alpha|\mathbf{00}\rangle + \beta|\mathbf{01}\rangle + \gamma|\mathbf{10}\rangle + \delta|\mathbf{11}\rangle \arrow[dll, "\mathbf{0}"', "|\alpha|^2+|\beta|^2" near start] \arrow[drr, "\mathbf{1}", "|\gamma|^2+|\delta|^2"' near start] & & & \\
 & \alpha|\mathbf{00}\rangle + \beta|\mathbf{01}\rangle \arrow[dl, "\mathbf{0}"', "\frac{|\alpha|^2}{|\alpha|^2+|\beta|^2}"] \arrow[dr, "\mathbf{1}", "\frac{|\beta|^2}{|\alpha|^2+|\beta|^2}"'] & & & & \gamma|\mathbf{10}\rangle + \delta|\mathbf{11}\rangle \arrow[dl, "\mathbf{0}"', "\frac{|\gamma|^2}{|\gamma|^2+|\delta|^2}"] \arrow[dr, "\mathbf{1}", "\frac{|\delta|^2}{|\gamma|^2+|\delta|^2}"'] & \\
\alpha|\mathbf{00}\rangle & & \beta|\mathbf{01}\rangle & & \gamma|\mathbf{10}\rangle & & \delta|\mathbf{11}\rangle
\end{tikzcd}
$$

Note: Normalization convention.

Pure vs. mixed states

A mixed state is a (classical) probability distribution on quantum states.

Ad hoc notation:

$$\frac{1}{2}\left\{\begin{pmatrix} \alpha \\ \beta \end{pmatrix}\right\} + \frac{1}{2}\left\{\begin{pmatrix} \alpha' \\ \beta' \end{pmatrix}\right\}$$

Note: A mixed state is a description of our *knowledge* of a state. An actual closed quantum system is always in a (possibly unknown) pure state.

Density matrices (von Neumann)

Represent the pure state $v = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \in \mathbb{C}^2$ by the matrix

$$vv^* = \begin{pmatrix} \alpha\bar{\alpha} & \alpha\bar{\beta} \\ \beta\bar{\alpha} & \beta\bar{\beta} \end{pmatrix} \in \mathbb{C}^{2\times 2}.$$

Represent the mixed state $\lambda_1 \{v_1\} + \ldots + \lambda_n \{v_n\}$ by

$$\lambda_1 v_1 v_1^* + \ldots + \lambda_n v_n v_n^*.$$

This representation is not one-to-one, e.g.

$$\frac{1}{2}\left\{\begin{pmatrix} 1 \\ 0 \end{pmatrix}\right\} + \frac{1}{2}\left\{\begin{pmatrix} 0 \\ 1 \end{pmatrix}\right\} = \frac{1}{2}\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + \frac{1}{2}\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} .5 & 0 \\ 0 & .5 \end{pmatrix}$$

$$\frac{1}{2}\left\{\frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix}\right\} + \frac{1}{2}\left\{\frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -1 \end{pmatrix}\right\} = \frac{1}{2}\begin{pmatrix} .5 & .5 \\ .5 & .5 \end{pmatrix} + \frac{1}{2}\begin{pmatrix} .5 & -.5 \\ -.5 & .5 \end{pmatrix} = \begin{pmatrix} .5 & 0 \\ 0 & .5 \end{pmatrix}$$

But these two mixed states are indistinguishable.

Quantum operations on density matrices

Unitary:

$$v \mapsto Uv \qquad\qquad vv^* \mapsto Uvv^*U^* \qquad\qquad A \mapsto UAU^*$$

Measurement:

$$\begin{tikzcd}[column sep=small]
& \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \arrow[dl, "|\alpha|^2"', "\mathbf{0}"] \arrow[dr, "|\beta|^2", "\mathbf{1}"'] & \\
\begin{pmatrix} \alpha \\ 0 \end{pmatrix} & & \begin{pmatrix} 0 \\ \beta \end{pmatrix}
\end{tikzcd}
\qquad
\begin{tikzcd}[column sep=small]
& \begin{pmatrix} \alpha\bar\alpha & \alpha\bar\beta \\ \beta\bar\alpha & \beta\bar\beta \end{pmatrix} \arrow[dl, "\alpha\bar\alpha"', "\mathbf{0}"] \arrow[dr, "\beta\bar\beta", "\mathbf{1}"'] & \\
\begin{pmatrix} \alpha\bar\alpha & 0 \\ 0 & 0 \end{pmatrix} & & \begin{pmatrix} 0 & 0 \\ 0 & \beta\bar\beta \end{pmatrix}
\end{tikzcd}
\qquad
\begin{tikzcd}[column sep=small]
& \begin{pmatrix} a & b \\ c & d \end{pmatrix} \arrow[dl, "a"', "\mathbf{0}"] \arrow[dr, "d", "\mathbf{1}"'] & \\
\begin{pmatrix} a & 0 \\ 0 & 0 \end{pmatrix} & & \begin{pmatrix} 0 & 0 \\ 0 & d \end{pmatrix}
\end{tikzcd}$$

A complete partial order of density matrices

Let $D_n = \{A \in \mathbb{C}^{n\times n} \mid A$ is positive hermitian and $\mathrm{tr}\, A \leq 1\}$.

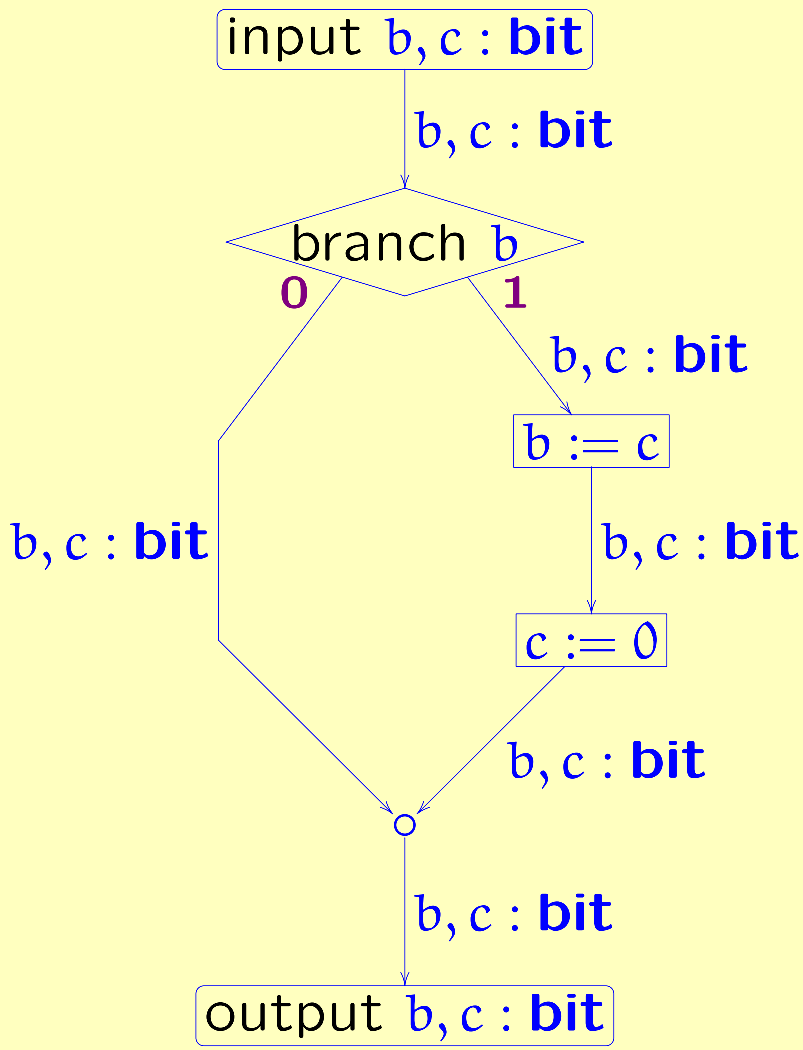
Definition. We write $A \sqsubseteq B$ if $B - A$ is positive.

Theorem. The density matrices form a *complete partial order* under $\sqsubseteq$.

- $A \sqsubseteq A$
- $A \sqsubseteq B$ and $B \sqsubseteq A \Rightarrow A = B$
- $A \sqsubseteq B$ and $B \sqsubseteq C \Rightarrow A \sqsubseteq C$
- every increasing sequence $A_1 \sqsubseteq A_2 \sqsubseteq \ldots$ has a least upper bound

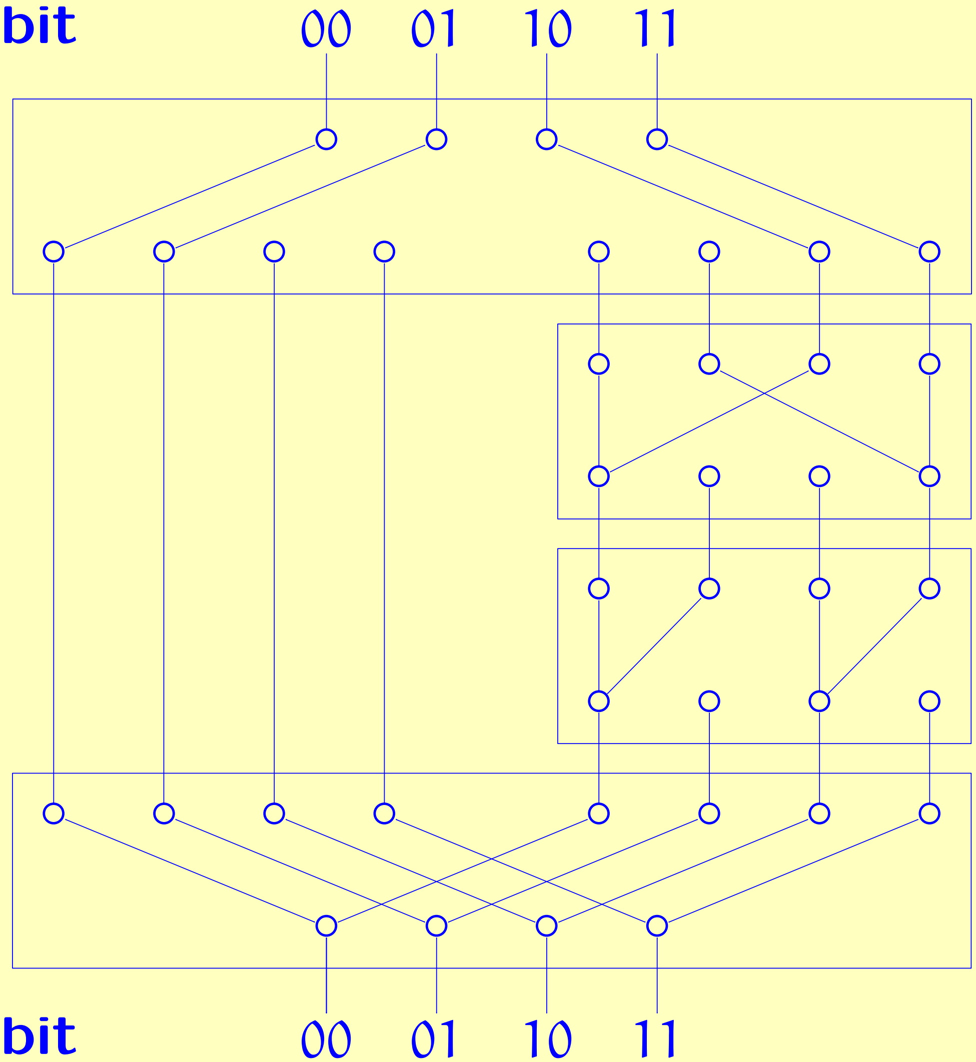
Part II: The Flow Chart Language

First: the classical case. A simple classical flow chart



Classical flow chart, with boolean variables expanded

input b, c : **bit**



$(*$ branch b $*)$

$(*\ b := c\ *)$

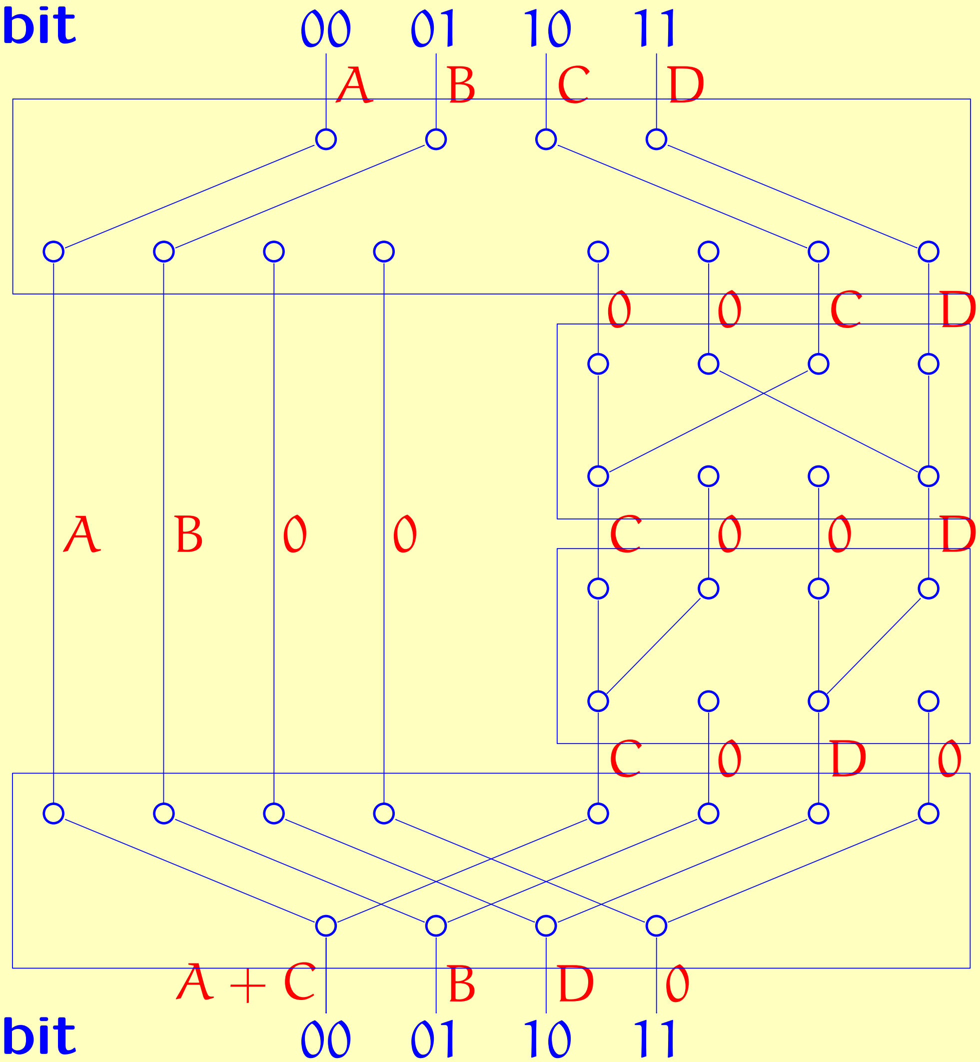
$(*\ c := 0\ *)$

$(*$ merge $*)$

output b, c : **bit**

Classical flow chart, with boolean variables expanded

input b, c : **bit**



(∗ branch b ∗)

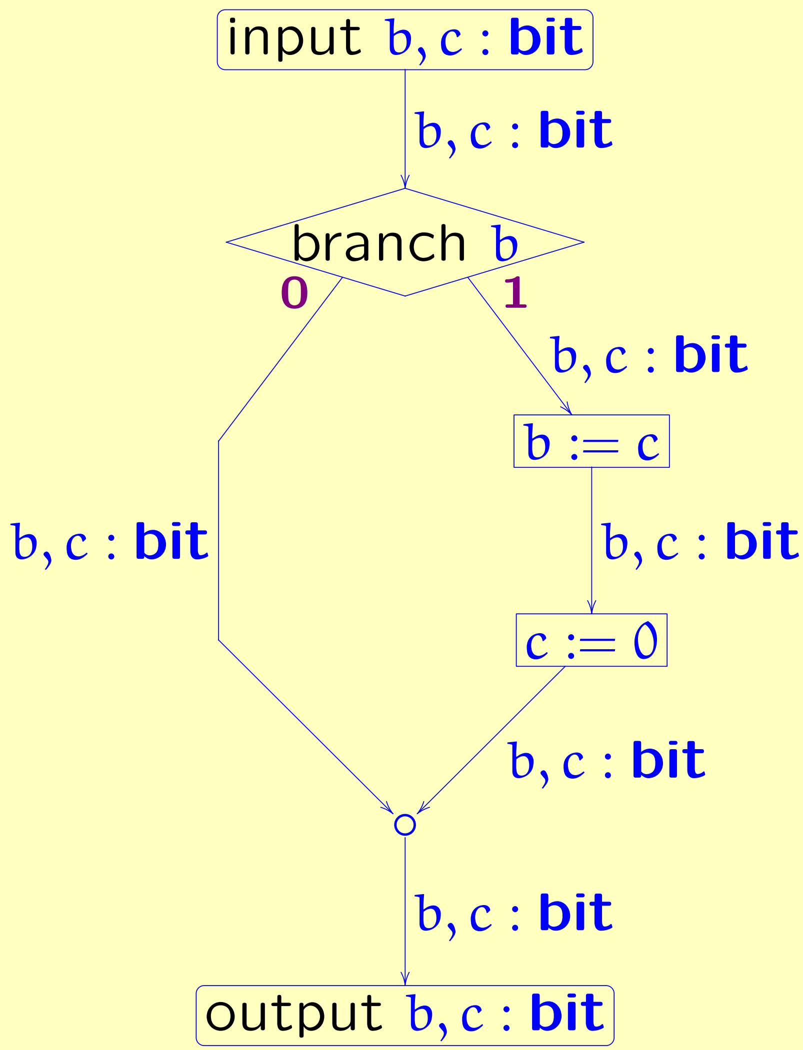
(∗ $b := c$ ∗)

(∗ $c := 0$ ∗)

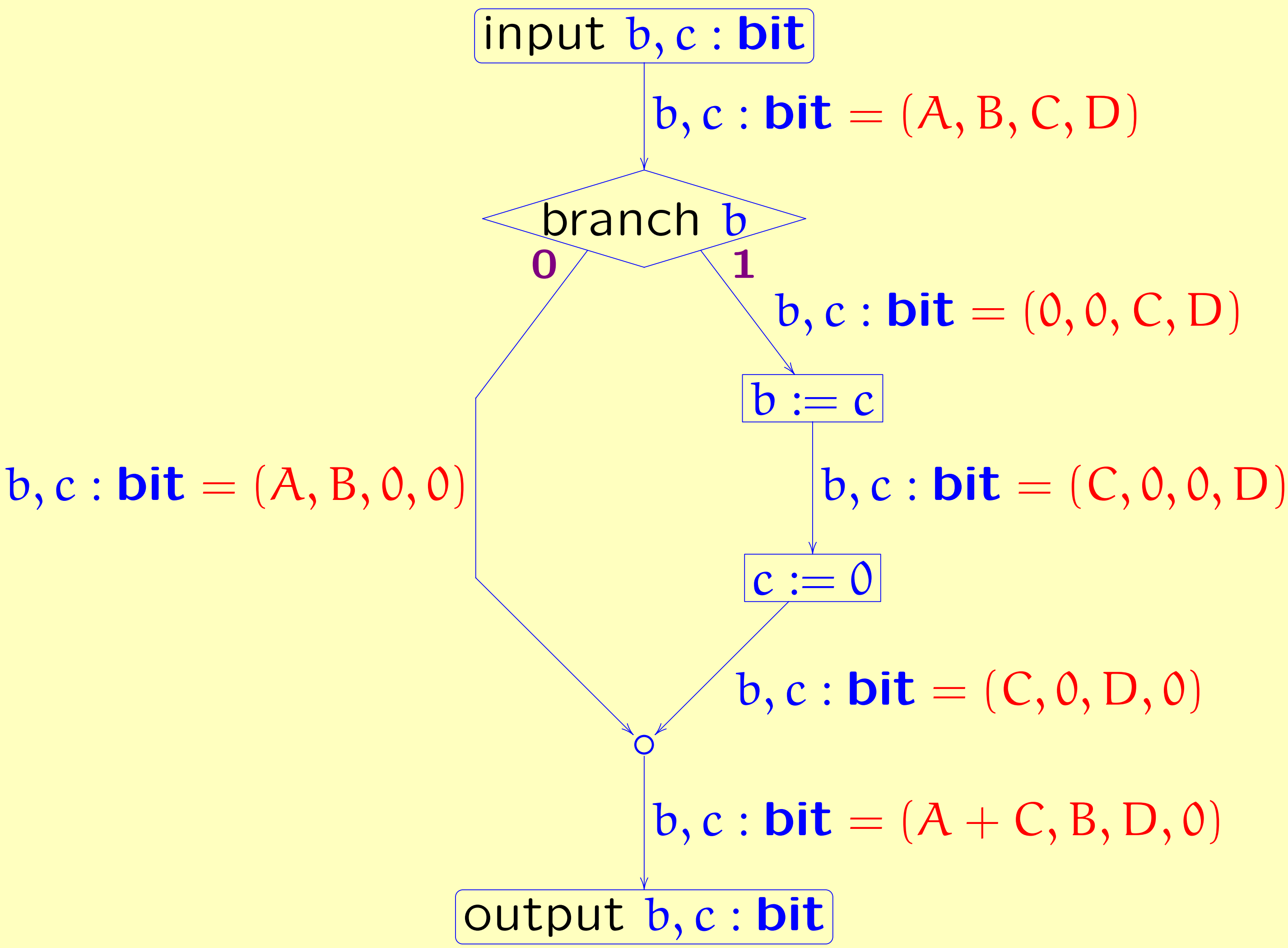
(∗ merge ∗)

output b, c : **bit**

A simple classical flow chart

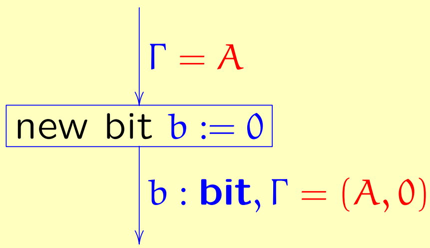


A simple classical flow chart

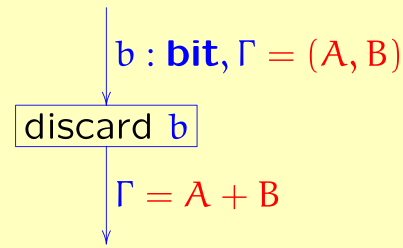


Summary of classical flow chart components

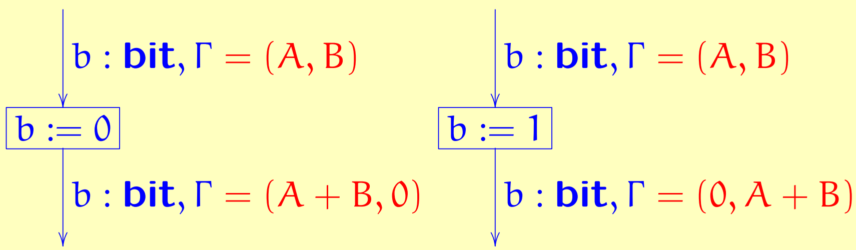
Allocate bit:



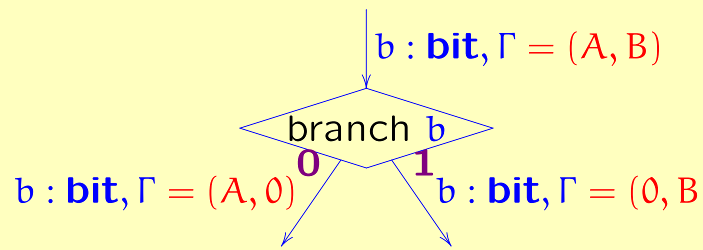
Discard bit:



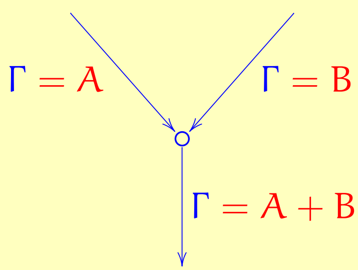
Assignment:



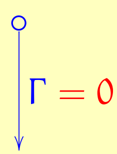
Branching:



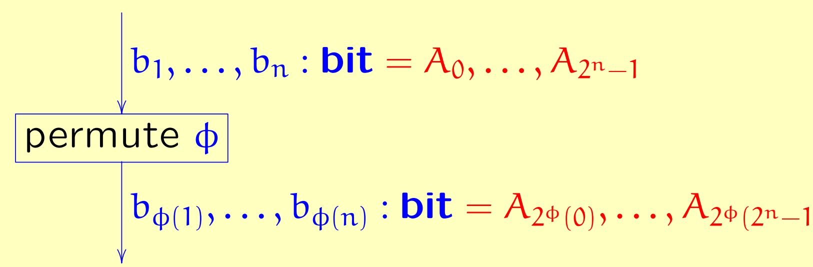
Merge:



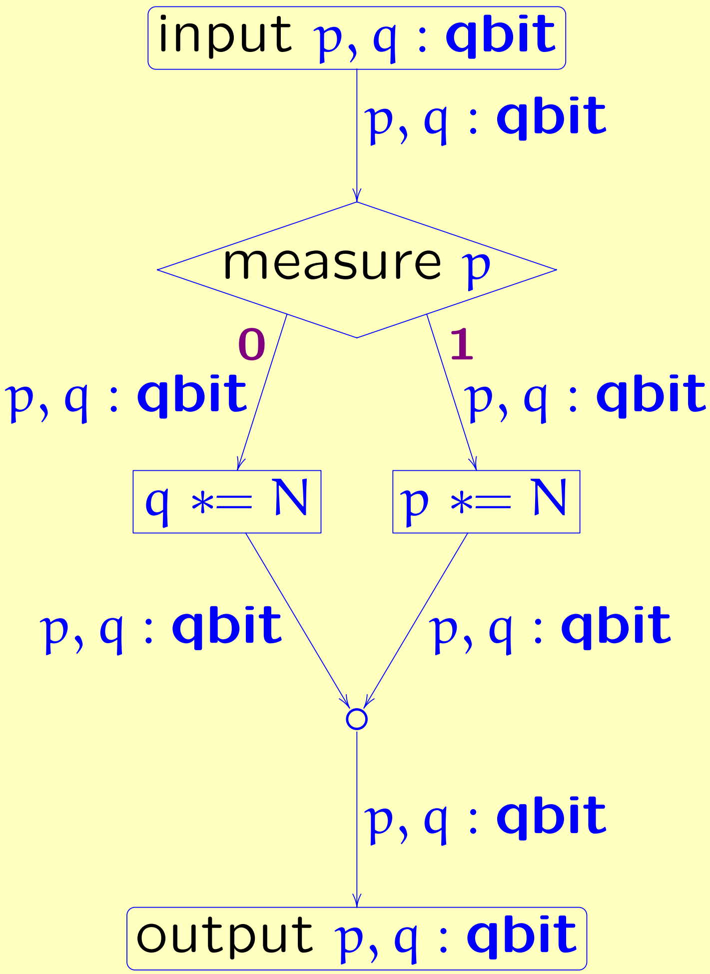
Initial:



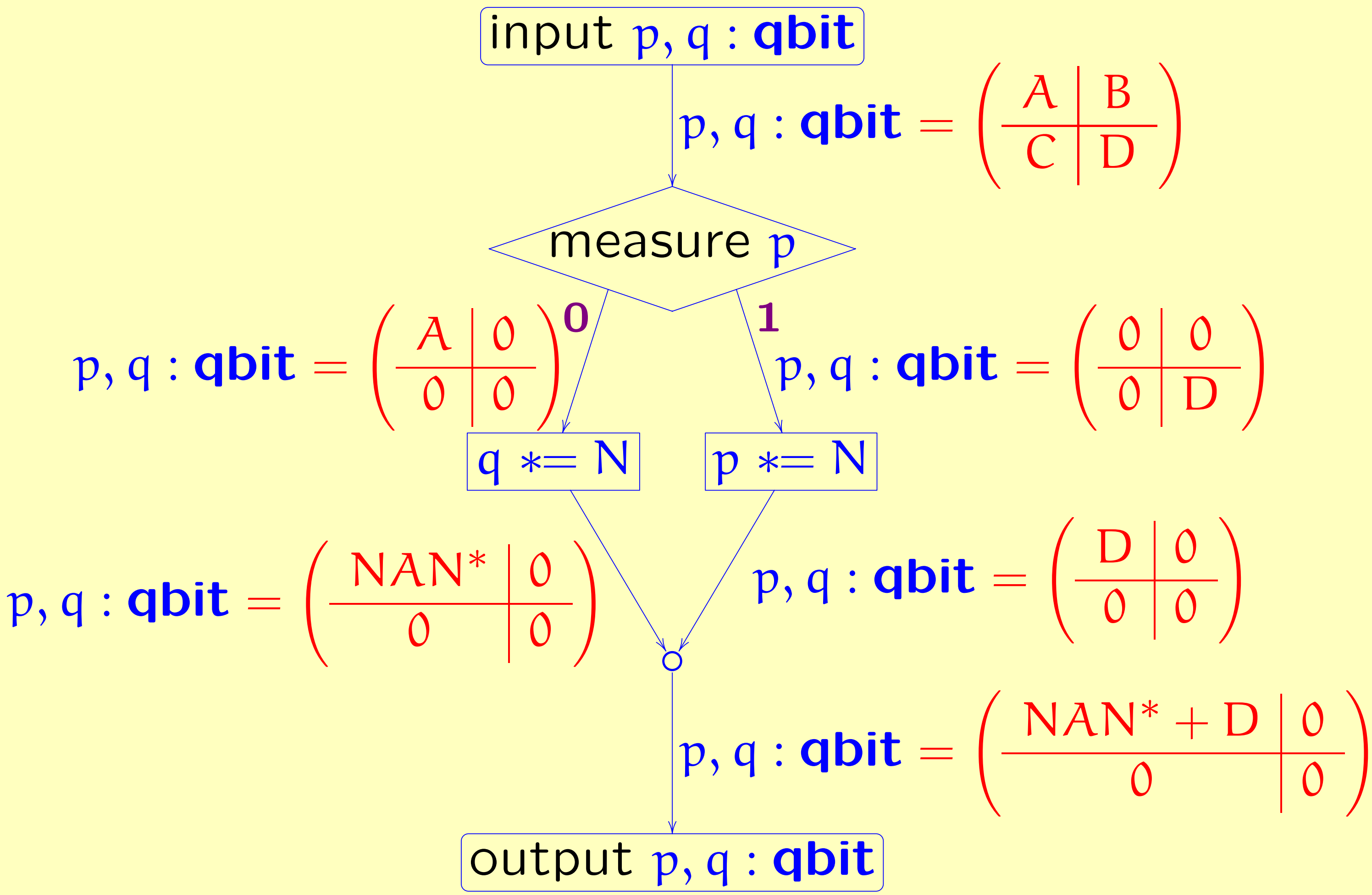
Permutation:



The quantum case: A simple quantum flow chart



A simple quantum flow chart



Summary of quantum flow chart components

Allocate qbit:

$\Gamma = A$

new qbit $q := 0$

$q : \textbf{qbit}, \Gamma = \left(\begin{array}{c|c} A & 0 \\ \hline 0 & 0 \end{array}\right)$

Discard qbit:

$q : \textbf{qbit}, \Gamma = \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)$

discard q

$\Gamma = A + D$

Unitary transformation:

$\bar{q} : \textbf{qbit}, \Gamma = A$

$\bar{q} \mathrel{*}= S$

$\bar{q} : \textbf{qbit}, \Gamma = (S \otimes I)A(S \otimes I)^*$

Measurement:

$q : \textbf{qbit}, \Gamma = \left(\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right)$

measure q

$\mathbf{0}$: $q : \textbf{qbit}, \Gamma = \left(\begin{array}{c|c} A & 0 \\ \hline 0 & 0 \end{array}\right)$

$\mathbf{1}$: $q : \textbf{qbit}, \Gamma = \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & D \end{array}\right)$

Merge:

$\Gamma = A$ $\quad$ $\Gamma = B$

$\Gamma = A + B$

Initial:

$\Gamma = 0$

Permutation:

$q_1, \ldots, q_n : \textbf{qbit} = (a_{ij})_{ij}$

permute ϕ

$q_{\phi(1)}, \ldots, q_{\phi(n)} : \textbf{qbit} = (a_{2^\phi(i),2^\phi(j)})_{ij}$

Part III: Quantum Lambda Calculus

With Benoît Valiron.

A quantum lambda calculus [Selinger,Valiron04]

- Quantum data is subject to *linearity constraints*. Need to avoid terms that lead to runtime errors such as

$$\mathsf{let}\ q = \mathsf{new\_qbit}()\ \mathsf{in}\ (\lambda x.H(x,x))\,q.$$

- Bits are always duplicable, qubits are never duplicable. What about functions?

- Consider

$$\begin{array}{rcl} q{:}\mathbf{qbit} & \vdash & \lambda p.p \ :\ \mathbf{qbit} \to \mathbf{qbit} \\ q{:}\mathbf{qbit} & \vdash & \lambda p.q \ :\ \mathbf{qbit} \to \mathbf{qbit} \end{array}$$

  Both closures have type $\mathbf{qbit} \to \mathbf{qbit}$, but only the first one is duplicable.

- Solution: type system based on linear logic.

Linear type system [Selinger,Valiron04]

Types: $A, B ::= \mathbf{qbit} \;\big|\; !A \;\big|\; A \multimap B \;\big|\; 1 \;\big|\; A \otimes B \;\big|\; A \oplus B$.

Convention $\mathbf{bit} := 1 \oplus 1$.

Subtyping: $!A <: A$.

Main typing rules:

$$\frac{\Delta, x{:}A \rhd M : B}{\Delta \rhd \lambda x.M : A \multimap B}$$

$$\frac{!\Delta, x{:}A \rhd M : B}{!\Delta \rhd \lambda x.M : !(A \multimap B)}$$

Complete typing rules:

$$\frac{A <: B}{\Delta, x{:}A \rhd x : B}\ (ax_1) \qquad \frac{!A_c <: B}{\Delta \rhd c : B}\ (ax_2)$$

$$\frac{\Delta \rhd M : !^n A}{\Delta \rhd \mathit{inj}_l(M) : !^n(A \oplus B)}\ (\oplus.I_1) \qquad \frac{\Delta \rhd N : !^n B}{\Delta \rhd \mathit{inj}_r(N) : !^n(A \oplus B)}\ (\oplus.I_2)$$

$$\frac{!\Delta, \Gamma_1 \rhd P : !^n(A \oplus B) \qquad \begin{array}{c} !\Delta, \Gamma_2, x : !^n A \rhd M : C \\ !\Delta, \Gamma_2, y : !^n B \rhd N : C \end{array}}{\Gamma_1, \Gamma_2, !\Delta \rhd \mathit{match}\ P\ \mathit{with}\ (x \mapsto M \mid y \mapsto N) : C}\ (\oplus.E)$$

$$\frac{\Gamma_1, !\Delta \rhd M : A \multimap B \qquad \Gamma_2, !\Delta \rhd N : A}{\Gamma_1, \Gamma_2, !\Delta \rhd MN : B}\ (\mathit{app})$$

If $FV(M) \cap |\Gamma| = \emptyset$:

$$\frac{x{:}A, \Delta \rhd M : B}{\Delta \rhd \lambda x.M : A \multimap B}\ (\lambda_1) \qquad \frac{\Gamma, !\Delta, x{:}A \rhd M : B}{\Gamma, !\Delta \rhd \lambda x.M : !^{n+1}(A \multimap B)}\ (\lambda_2)$$

$$\frac{!\Delta, \Gamma_1 \rhd M_1 : !^n A_1 \qquad !\Delta, \Gamma_2 \rhd M_2 : !^n A_2}{!\Delta, \Gamma_1, \Gamma_2 \rhd \langle M_1, M_2 \rangle : !^n(A_1 \otimes A_2)}\ (\otimes.I) \qquad \frac{}{\Delta \rhd * : !^n 1}\ (1)$$

$$\frac{!\Delta, \Gamma_1 \rhd M : !^n(A_1 \otimes A_2) \qquad !\Delta, \Gamma_2,\ x_1{:}!^n A_1,\ x_2{:}!^n A_2 \rhd N : A}{!\Delta, \Gamma_1, \Gamma_2 \rhd \mathrm{let}\ \langle x_1, x_2 \rangle = M\ \mathrm{in}\ N : A}\ (\otimes.E)$$

$$\frac{!\Delta, f : !(A \multimap B), x : A \rhd M : B \qquad !\Delta, \Gamma, f : !(A \multimap B) \rhd N : C}{!\Delta, \Gamma \rhd \mathit{let\ rec}\ f\ x = M\ \mathit{in}\ N : C}\ (\mathit{rec})$$

Properties of the type system

All the rules of intuitionistic linear logic are valid, except for the general promotion rule:

$$\frac{!\Delta \rhd M : A}{!\Delta \rhd M : !A.}$$

We do have the promotion rule for *values*:

$$\frac{!\Delta \rhd V : A}{!\Delta \rhd V : !A.}$$

Type inference: first do "intuitionistic" type inference, then find a "linear decoration".

Completeness

Quantum lambda calculus (with list types and recursion) is complete for quantum computation. Every algorithm can be expressed *in principle*.

Part IV: Quipper

Quipper developers: Richard Eisenberg, Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, Benoît Valiron.

Design goals

Quantum lambda calculus is *too low-level*. Algorithms in the quantum literature are described in terms of *meta-operations*:

- Start with a classical function;
- turn it into a circuit;
- make it reversible;
- apply a transformation (e.g. amplitude amplification);
- etc.

Quipper: extend quantum lambda calculus with the ability to *build and manipulate quantum circuits as first-class objects*.

Quipper's initial implementation

Implemented as a deeply embedded EDSL in Haskell.

Reasons:

- Haskell provides very good support for *higher-order*, *polymorphic*, and *overloaded* functions.
- Both Haskell and Quipper are strongly typed, functional programming languages, and as such, are a relatively good fit for each other.

Trade-offs:

- Haskell lacks two features that would be useful to Quipper: *linear types* and *dependent types*. We must live with checking certain properties at run-time that could be checked by the type-checker in a dedicated language.

Quipper contains a powerful circuit description language

- In our experience, 99 percent of the quantum programmer's task is *"constructing the circuit"*, and 1 percent is *"running the circuit"*.

- Quipper separates the *description* of quantum operations from *what to do with them*. E.g.: a given quantum function could be:
  - executed right away;
  - stored for later execution; or
  - stored to be transformed or analyzed.

- Many tasks in algorithm construction require manipulations at the *circuit level*, rather than the *gate level*. For example:
  - reversing;
  - iteration (e.g. Trotterization; amplitude amplification);
  - construction of classical oracles and ancilla management;
  - whole-circuit optimization

The two run-times

As a circuit description language, Quipper shares many features with *hardware description languages*. In particular, it has *three distinct phases of execution*:

1. **Compile time.**
   Subject to: *compile time parameters*
   Error detection: most programming errors detected.

2. **Circuit generation time (“synthesizer”).**
   Subject to: *circuit parameters*
   Error detection: ideally none (or some run-time errors).

3. **Circuit execution time.**
   Subject to: *circuit inputs*
   Error detection: decoherence errors, physical errors.

The two run-times, continued

The distinction between *parameters* and *inputs* requires special support in the type system. **Circuit inputs are not known at circuit generation time!**

In Quipper, this is done by having 3 basic types instead of the usual 2:

- **Bool**: a boolean *parameter*, known at circuit generation time;
- **Bit**: a boolean *input*, i.e., a boolean wire in a circuit;
- **Qubit**: a qubit *input*, i.e., a qubit wire in a circuit.

Moreover, circuit generation and execution may be interleaved (*"dynamic lifting"*).

The Quipper idiom

The basic idiom for writing a circuit in Quipper is:

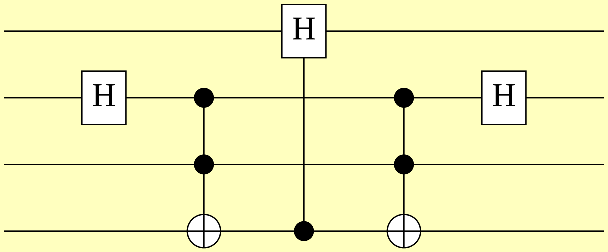
```
example :: (Qubit, Qubit, Qubit) -> Circ (Qubit, Qubit, Qubit)
example (a, b, c) = do
  <<gate1>>
  <<gate2>>
  <<gate3>>
  return (a, b, c)
```

Note: in Quipper, as in Quantum Lambda Calculus, quantum operations are viewed as functions. However, the *Circ* monad is used to assemble a data structure.

A first example

The code on the left is a small, but complete, Quipper program. When it is compiled and run, it outputs the circuit shown on the right.

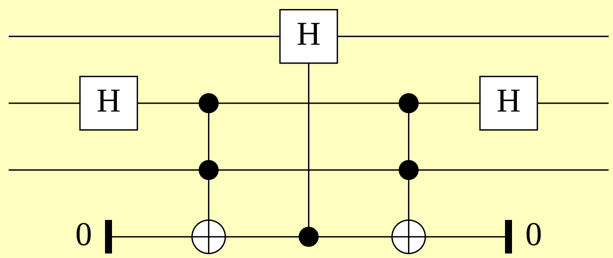
```
import Quipper

example1 (q, a, b, c) = do
  hadamard a
  qnot_at c ‘controlled‘ [a, b]
  hadamard q ‘controlled‘ [c]
  qnot_at c ‘controlled‘ [a, b]
  hadamard a
  return (q, a, b, c)
```



Scoped ancillas

Let us modify the previous example so that the qubit c is a local ancilla. This is done with the **`with_ancilla`** operator. This operator is followed by a nested “do” block. Note that Quipper uses indentation to figure out the end of a “do” block.

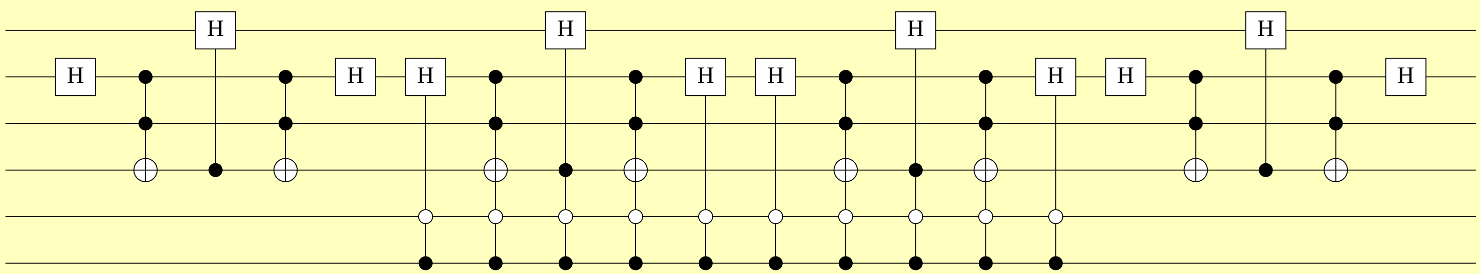
```
import Quipper

example2 (q, a, b) = do
  hadamard a
  with_ancilla $ \c -> do
    qnot_at c ‘controlled‘ [a, b]
    hadamard q ‘controlled‘ [c]
    qnot_at c ‘controlled‘ [a, b]
  hadamard a
  return (q, a, b)
```



Blockwise controls

Any circuit previously defined can be used as a subroutine. Also, the `with_controls` operator can be used to specify that an entire block of gates should be controlled:

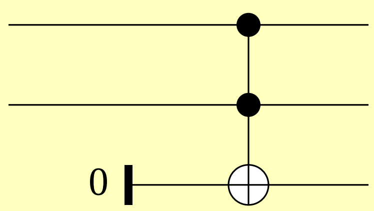
```
example3 (q, a, b, c, d, e) = do
  example1 (q, a, b, c)
  with_controls (d .=. 0 .&&. e .=. 1) $ do
    example1 (q, a, b, c)
    example1 (q, a, b, c)
  example1 (q, a, b, c)
```



Recursion

Let us consider an implementation of a classical "and" gate. It inputs two qubits, and returns a new ancilla qubit initialized to the "and" of the two input qubits:

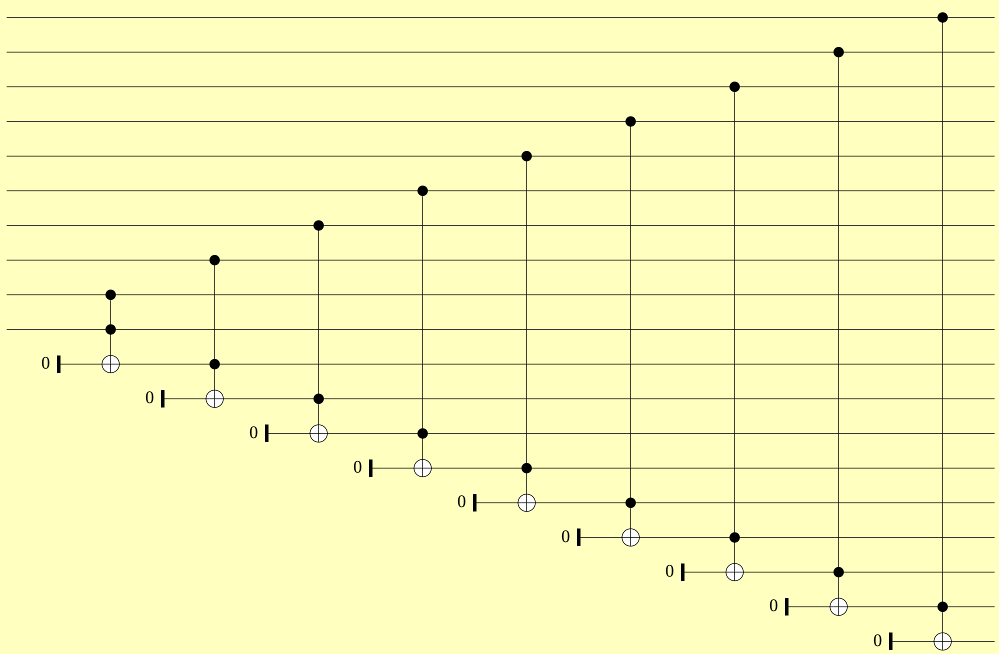
```
and_gate :: (Qubit, Qubit) -> Circ (Qubit)
and_gate (a, b) = do
  c <- qinit False
  qnot_at c `controlled` [a, b]
  return c
```



Recursion, continued

We now program a function that computes the “and” of a list of qubits. Note that the length of the list is a *parameter*, but the qubits themselves are *inputs*.

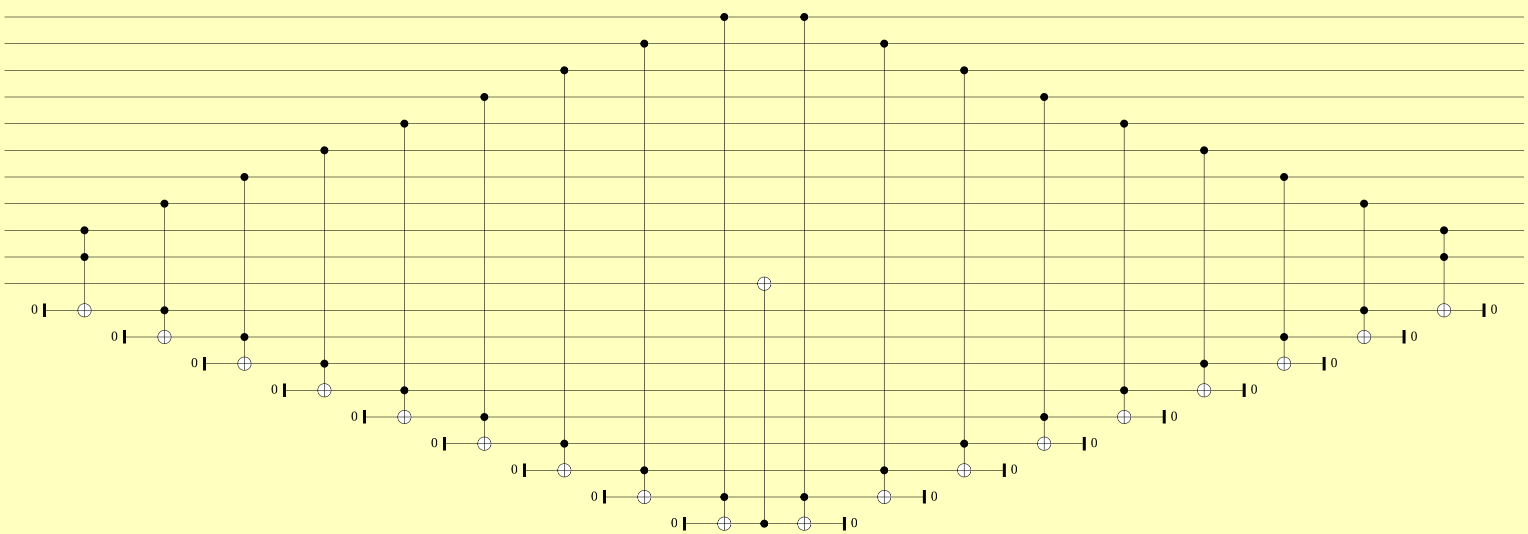
```
and_list :: [Qubit] -> Circ Qubit
and_list [] = do
  c <- qinit True
  return c
and_list [q] = do
  return q
and_list (q:t) = do
  d <- and_list t
  e <- and_gate (d, q)
  return e
```



Generic classical-to-reversible operator with ancilla uncomputation

Quipper provides a general operator `classical_to_reversible`, which turns any classical circuit into a reversible circuit by uncomputing all the "garbage" ancillas. When applying this to the function from the previous slide, we get:

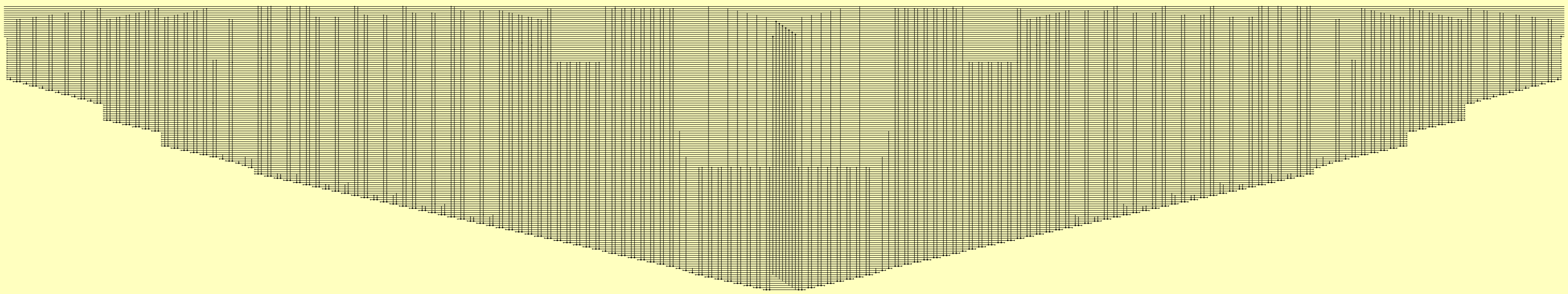
```
and_rev :: ([Qubit], Qubit) -> Circ ([Qubit], Qubit)
and_rev = classical_to_reversible and_list
```



Automatic oracle generation from classical code

Quipper can generate circuits from ordinary classical functional programs, via *Template Haskell* and a preprocessor.

```
build_circuit
v_function :: BoolParam -> BoolParam -> Boollist -> Boollist -> Node -> (Bool,No
v_function c_hi c_lo f g a =
  let aa = snd a in
  let cbc_hi = newBool c_hi `bool_xor` level_parity aa in
  let cbc_lo = newBool c_lo in
    if (not (is_root aa) && cbc_hi && not (cbc_lo `bool_xor` (last aa)))
    then (False, parent a)
    else
      let res = child f g a cbc_lo in
        (is_zero aa || cbc_hi, res)
```



Part V: Issues for type system design

Inadequacy of host language

- Haskell has no *linear types*. Therefore, errors like

```
(a,b) <- controlled_not (c,c)
```

  can only be caught at runtime.

- Haskell has no *dependent types*. Variable size circuits can be reprented as

```
circuit :: [Qubit] -> Circ [Qubit]
```

  However, when *reversing* such a circuit, the type system cannot know the number of inputs of the reversed circuit. That is because the length of the list is a *parameter* but here treated as an *input*. Dependent types would solve this problem easily.

Some issues for type system design

- Reversibility tracking (not all circuits are reversible).
- Support for imperative syntax. Writing gates in purely functional style is okay:

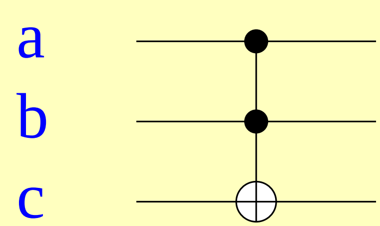
  ```
  (a,b) <- gate (a,b).
  ```

  However, for higher-order combinators, this turns very ugly:

  ```
  (a,b) <- (while <<condition>> do \(a,b) ->
     <<body>>
     return (a,b)
  ) (a,b)
  ```

Some issues for type system design, continued

- Weak linearity. Consider the classical Toffoli gate



Linearity requires that $a \neq c$ and $b \neq c$.

On the other hand, it can be perfectly reasonable (and desirable) to allow $a = b$.

Moreover, a and b are immutable.

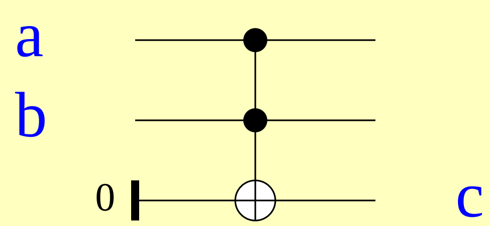
In classical circuit synthesis, to ensure well-formed circuits, one should keep track of all of these properties.

Some issues for type system design, continued

- Automatic garbage management. A classical boolean function such as

```
let c = and a b
```

will be synthesized to a circuit such as this:



Note that a and b are outputs of the circuit, but not of the function. They are "garbage", and must potentially be uncomputed later. We need a type system to automatically track such garbage.

The end.