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 \( A^* = (a_{ji})_{ij} \), trace \( \text{tr} A = \sum_i a_{ii} \), norm \( \|A\|^2 = \sum_{ij} |a_{ij}|^2 \).

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

- 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: $\alpha|0\rangle + \beta|1\rangle$ (superposition of $|0\rangle$ and $|1\rangle$).

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

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

- otherwise entangled.
Lexicographic convention

Identify the basis states $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$ 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.

\[
\begin{pmatrix}
\alpha \\
\beta \\
\gamma \\
\delta 
\end{pmatrix} = \alpha |00\rangle + \beta |01\rangle + \gamma |10\rangle + \delta |11\rangle.
\]
Quantum computation: Operations

- unitary transformation

- measurement
Some standard unitary gates

Unary:

\[ N = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad N_c = \begin{pmatrix} I & 0 \\ 0 & N \end{pmatrix}, \]

\[ H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad H_c = \begin{pmatrix} I & 0 \\ 0 & H \end{pmatrix}, \]

\[ V = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \quad V_c = \begin{pmatrix} I & 0 \\ 0 & V \end{pmatrix}, \]

\[ W = \begin{pmatrix} 1 & 0 \\ 0 & \sqrt{i} \end{pmatrix}, \quad W_c = \begin{pmatrix} I & 0 \\ 0 & W \end{pmatrix}, \]

Binary:

\[ X = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \]
Measurement

\[ \alpha |0\rangle + \beta |1\rangle \]

\[ |\alpha|^2 \quad 0 \quad 1 \quad |\beta|^2 \]

\[ \alpha |0\rangle \quad \beta |1\rangle \]
Two Measurements

\[ \alpha |00\rangle + \beta |01\rangle + \gamma |10\rangle + \delta |11\rangle \]

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

\[
vv^* = \begin{pmatrix} \alpha \bar{\alpha} & \alpha \bar{\beta} \\ \beta \bar{\alpha} & \beta \bar{\beta} \end{pmatrix} \in \mathbb{C}^{2\times2}.
\]

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:

\[ \rho \mapsto U \rho U^* \]

Measurement:

\[
\begin{pmatrix}
\alpha \\
\beta \\
\end{pmatrix}
\begin{pmatrix}
0 & 1 \\
|\alpha|^2 & |\beta|^2 \\
\end{pmatrix}
\begin{pmatrix}
\alpha \\
0 \\
\end{pmatrix}
\begin{pmatrix}
\bar{\alpha} \\
\beta \\
\end{pmatrix}
\begin{pmatrix}
0 & 1 \\
\alpha \bar{\alpha} & \beta \bar{\beta} \\
\end{pmatrix}
\begin{pmatrix}
0 & 0 \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
a & b \\
c & d \\
\end{pmatrix}
\begin{pmatrix}
a \bar{\alpha} & \beta \bar{\beta} \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
a & 0 \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
a & b \\
c & d \\
\end{pmatrix}
\begin{pmatrix}
a & 0 \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
a & 0 \\
0 & 0 \\
\end{pmatrix}
\]

\[ \begin{pmatrix}
\alpha \\
\beta \\
\end{pmatrix}
\begin{pmatrix}
\alpha \bar{\alpha} & \alpha \bar{\beta} \\
\beta \bar{\alpha} & \beta \bar{\beta} \\
\end{pmatrix}
\begin{pmatrix}
\alpha \bar{\alpha} & 0 \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
\alpha \bar{\alpha} & 0 \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
\alpha \bar{\alpha} & 0 \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
a & 0 \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
a & 0 \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
a & 0 \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
a & 0 \\
0 & 0 \\
\end{pmatrix}
\]

14
A complete partial order of density matrices

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

**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

```
input b, c : bit

b, c : bit
branch b
0 1
b, c : bit
b := c
b, c : bit
c := 0
b, c : bit
b, c : bit
output b, c : bit
```
Classical flow chart, with boolean variables expanded

input $b, c : \textbf{bit}$

00 01 10 11

(* branch $b$ *)

(* $b := c$ *)

(* $c := 0$ *)

(* merge *)

output $b, c : \textbf{bit}$

00 01 10 11

18
Classical flow chart, with boolean variables expanded

input \( b, c : \text{bit} \)

\[
\begin{array}{cccc}
00 & 01 & 10 & 11 \\
\text{A} & \text{B} & \text{C} & \text{D} \\
\end{array}
\]

(output \( b, c : \text{bit} \))

\[
\begin{array}{cccc}
00 & 01 & 10 & 11 \\
\text{A + C} & \text{B} & \text{D} & 0 \\
\end{array}
\]

(* branch \( b \) *)

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

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

(* merge *)
A simple classical flow chart

```
input b, c : bit

branch b

0

b, c : bit

b := c

b, c : bit
c := 0

b, c : bit

output b, c : bit
```
A simple classical flow chart

input $b, c : \text{bit}$

$b, c : \text{bit} = (A, B, C, D)$

branch $b$

1

$b, c : \text{bit} = (0, 0, C, D)$

$b := c$

$b, c : \text{bit} = (A, B, 0, 0)$

$c := 0$

$b, c : \text{bit} = (C, 0, 0, D)$

$b, c : \text{bit} = (C, 0, D, 0)$

output $b, c : \text{bit}$

$b, c : \text{bit} = (A + C, B, D, 0)$
Summary of classical flow chart components

Allocate bit:
\[ \Gamma = A \]
\[ \text{new bit } b := 0 \]
\[ b : \text{bit}, \Gamma = (A, 0) \]

Discard bit:
\[ b : \text{bit}, \Gamma = (A, B) \]
\[ \text{discard } b \]
\[ \Gamma = A + B \]

Assignment:
\[ b : \text{bit}, \Gamma = (A, B) \]
\[ b := 0 \]
\[ b : \text{bit}, \Gamma = (A + B, 0) \]
\[ b := 1 \]
\[ b : \text{bit}, \Gamma = (0, A + B) \]

Branching:
\[ b : \text{bit}, \Gamma = (A, B) \]
\[ \text{branch } b \]
\[ b : \text{bit}, \Gamma = (A, 0) \]
\[ b : \text{bit}, \Gamma = (0, B) \]

Merge:
\[ \Gamma = A \]
\[ \Gamma = B \]
\[ \Gamma = A + B \]
\[ \Gamma = 0 \]

Initial:
\[ b_1, \ldots, b_n : \text{bit} = A_0, \ldots, A_{2^n-1} \]
\[ \text{permute } \phi \]
\[ b_{\phi(1)}, \ldots, b_{\phi(n)} : \text{bit} = A_{2^\phi(0)}, \ldots, A_{2^\phi(2^n-1)} \]
The quantum case: A simple quantum flow chart

```
input p, q : qbit

measure p

p : qbit 0
q *= N
p : qbit

p : qbit 1
p *= N
p : qbit

output p, q : qbit
```
A simple quantum flow chart

\[ \text{input } p, q : \text{qbit} \]

\[ p, q : \text{qbit} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \]

measure p

\[ p, q : \text{qbit} = \begin{pmatrix} A & 0 \\ 0 & 0 \end{pmatrix} \]

\[ p, q : \text{qbit} = \begin{pmatrix} 0 & 0 \\ 0 & D \end{pmatrix} \]

\[ q *\equiv N \]

\[ p *\equiv N \]

\[ p, q : \text{qbit} = \begin{pmatrix} \text{NAN}\star & 0 \\ 0 & 0 \end{pmatrix} \]

\[ p, q : \text{qbit} = \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \]

\[ p, q : \text{qbit} = \begin{pmatrix} \text{NAN}\star + D & 0 \\ 0 & 0 \end{pmatrix} \]

output p, q : qbit
Summary of quantum flow chart components

Allocate qbit:
\[
\begin{align*}
\Gamma &= A \\
\text{new qbit } q := 0 \\
q : \text{qbit}, \Gamma &= \begin{pmatrix} A & 0 \\ 0 & 0 \end{pmatrix}
\end{align*}
\]

Discard qbit:
\[
\begin{align*}
q : \text{qbit}, \Gamma &= \begin{pmatrix} A & B \\ C & D \end{pmatrix} \\
discard q \\
\Gamma &= A + D
\end{align*}
\]

Unitary transformation:
\[
\begin{align*}
\tilde{q} : \text{qbit}, \Gamma &= A \\
\tilde{q} &= S \\
\tilde{q} : \text{qbit}, \Gamma &= (S \otimes I)A(S \otimes I)^* 
\end{align*}
\]

Measurement:
\[
\begin{align*}
q : \text{qbit}, \Gamma &= \begin{pmatrix} A & 0 \\ 0 & 0 \end{pmatrix} \\
\text{measure } q \\
q : \text{qbit}, \Gamma &= \begin{pmatrix} 0 & 0 \\ 0 & D \end{pmatrix}
\end{align*}
\]

Merge:
\[
\begin{align*}
\Gamma &= A \\
\Gamma &= B \\
\Gamma &= A + B
\end{align*}
\]

Initial:
\[
\begin{align*}
\Gamma &= A \\
\Gamma &= B \\
\Gamma &= A + B \\
\Gamma &= 0
\end{align*}
\]

Permutation:
\[
\begin{align*}
q_1, \ldots, q_n : \text{qbit} &= (a_{ij})_{ij} \\
\text{permute } \phi \\
q_{\phi(1)}, \ldots, q_{\phi(n)} : \text{qbit} &= (a_{2\phi(i),2\phi(j)})_{ij}
\end{align*}
\]
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

  \[
  \text{let } q = \text{new}_\text{qbit}() \text{ in } (\lambda x. H(x, x)) q.
  \]

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

- Consider

  \[
  \begin{align*}
  q: \text{qbit} & \vdash \lambda p.p : \text{qbit} \rightarrow \text{qbit} \\
  q: \text{qbit} & \vdash \lambda p.q : \text{qbit} \rightarrow \text{qbit}
  \end{align*}
  \]

  Both closures have type \text{qbit} \rightarrow \text{qbit}, but only the first one is duplicable.

- Solution: type system based on linear logic.
Linear type system [Selinger, Valiron04]

Types: \( A, B ::= \text{qbit} \mid !A \mid A \rightarrow B \mid 1 \mid A \otimes B \mid A \oplus B. \)

Convention \( \text{bit} := 1 \oplus 1. \)

Subtyping: \( !A <: A. \)
Main typing rules:

\[
\frac{\Delta, x:A \triangleright M : B}{\Delta \triangleright \lambda x.M : A \rightarrow B}
\]

\[
\frac{!\Delta, x:A \triangleright M : B}{!\Delta \triangleright \lambda x.M : !(A \rightarrow B)}
\]
Complete typing rules:

\[
\frac{A \triangleleft: B}{\Delta, x:A \triangleright x : B} \quad (ax_1)
\]
\[
\frac{\Delta \triangleright c : B}{\Delta \triangleright c : B} \quad (ax_2)
\]
\[
\frac{\Delta \triangleright M : !^m A \quad \Delta \triangleright \text{inj}_1(M) : !^m (A \oplus B)}{\Delta \triangleright \text{inj}_1(M) : !^m (A \oplus B)} \quad (\oplus.I_1)
\]
\[
\frac{\Delta \triangleright N : !^m B \quad \Delta \triangleright \text{inj}_r(N) : !^m (A \oplus B)}{\Delta \triangleright \text{inj}_r(N) : !^m (A \oplus B)} \quad (\oplus.I_2)
\]
\[
\frac{\Delta \triangleright \text{match } P \text{ with } (x \mapsto M \mid y \mapsto N) : C}{\Gamma_1, \Gamma_2, !\Delta \triangleright \text{match } P \text{ with } (x \mapsto M \mid y \mapsto N) : C \quad (\oplus.E)}
\]
\[
\frac{\Gamma_1, !\Delta \triangleright M : A \rightarrow B \quad \Gamma_2, !\Delta \triangleright N : A}{\Gamma_1, \Gamma_2, !\Delta \triangleright MN : B \quad (\text{app})}
\]
\[
\begin{align*}
\text{If } & \text{ FV}(M) \cap |\Gamma| = \emptyset: \\
\frac{\Gamma, !\Delta, x:A \triangleright M : B}{\Gamma, !\Delta, x:A \triangleright M : B} & \quad (\lambda_1) \\
\frac{\Delta \triangleright \lambda x.\bar{M} : A \rightarrow B}{\Gamma, !\Delta \triangleright \lambda x.\bar{M} : !^{n+1}(A \rightarrow B)} & \quad (\lambda_2) \\
\end{align*}
\]
\[
\frac{\Delta \triangleright M_1 : !^n A_1 \quad \Delta \triangleright M_2 : !^n A_2}{\Delta \triangleright \langle M_1, M_2 \rangle : !^n (A_1 \otimes A_2)} \quad (\otimes.I)
\]
\[
\frac{\Delta \triangleright \langle M_1, M_2 \rangle : !^n (A_1 \otimes A_2)}{\Delta \triangleright \star : !^m 1} \quad (1)
\]
\[
\frac{\Delta \triangleright M : !^n (A_1 \otimes A_2) \quad \Delta \triangleright x_1 : !^n A_1, x_2 : !^n A_2 \triangleright N : A}{\Delta \triangleright \text{let } \langle x_1, x_2 \rangle = M \text{ in } N : A \quad (\otimes.E)}
\]
\[
\frac{\Delta \triangleright f : !(A \rightarrow B), x : A \triangleright M : B \quad \Delta, \Gamma, f : !(A \rightarrow B) \triangleright N : C}{\Delta, \Gamma \triangleright \text{let rec } f x = M \text{ in } N : C \quad (\text{rec})}
\]
Properties of the type system

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

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

We do have the promotion rule for \textit{values}:

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

\textbf{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 \textit{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:

```haskell
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.

```haskell
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.

```quipper
import Quipper

def 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)
```

---

![Diagram](image-url)
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:

```haskell
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)
    example1 (q, a, b, c)
```

![Diagram showing quantum circuit with blockwise controls]
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:

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

![Diagram of the control qubit in a circuit diagram](image)
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.

\[
\text{and\_list} :: [\text{Qubit}] \rightarrow \text{Circ Qubit}
\]

\[
\text{and\_list} \; [] \; = \; \text{do}
\]

\[
\text{c} \; \leftarrow \; \text{qinit} \; \text{True}
\]

\[
\text{return} \; \text{c}
\]

\[
\text{and\_list} \; [q] \; = \; \text{do}
\]

\[
\text{return} \; q
\]

\[
\text{and\_list} \; (q:t) \; = \; \text{do}
\]

\[
d \; \leftarrow \; \text{and\_list} \; t
\]

\[
e \; \leftarrow \; \text{and\_gate} \; (d, \; q)
\]

\[
\text{return} \; e
\]
Generic classical-to-reversible operator with ancilla uncomputation

Quipper provides a general operator \texttt{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:

\begin{verbatim}
and_rev :: ([Qubit], Qubit) -> Circ ([Qubit], Qubit)
and_rev = classical_to_reversible and_list
\end{verbatim}
Automatic oracle generation from classical code

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

```haskell
build_circuit
v_function :: BoolParam -> BoolParam -> Boollist -> Boollist -> Node -> (Bool,Node)
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

   \[(a,b) \leftarrow \text{controlled_not} (c,c)\]

   can only be caught at runtime.

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

   \[
   \text{circuit} :: [\text{Qubit}] \rightarrow \text{Circ} [\text{Qubit}]
   \]

   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:

  \[(a,b) \leftarrow \text{gate} (a,b).\]

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

\[(a,b) \leftarrow (\text{while } \langle\langle \text{condition} \rangle \rangle \text{ do } \langle\langle \text{body} \rangle \rangle \rightarrow \\
\langle\langle \text{return} (a,b) \rangle \rangle)(a,b)\]
Some issues for type system design, continued

• Weak linearity. Consider the classical Toffoli gate

```
  a -----
     |
  b ----
    |  
  c    
```

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

  \[
  \text{let } c = \text{and } a \ b
  \]

  will be synthesized to a circuit such as this:

  \[
  \begin{array}{c}
  a \\
  \downarrow \\
  b \\
  \downarrow \\
  0 \oplus \quad c
  \end{array}
  \]

  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.