Quantum Programming Languages

Benoît Valiron (CentraleSupélec / LMF)

Nov 2025

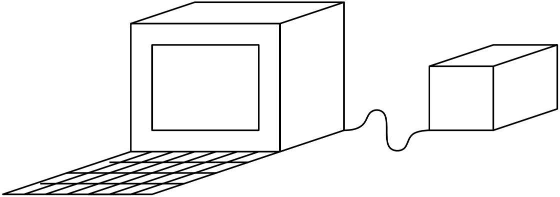
1st QCOMICAL School

Plan

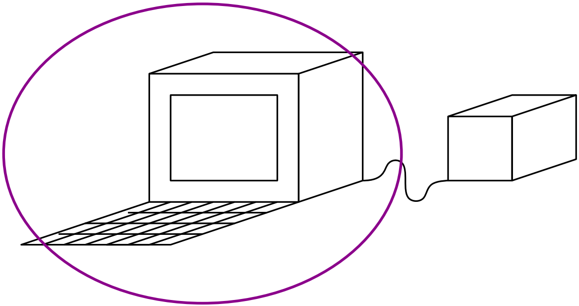
Plan

Plan

Model of Computation: Co-processor

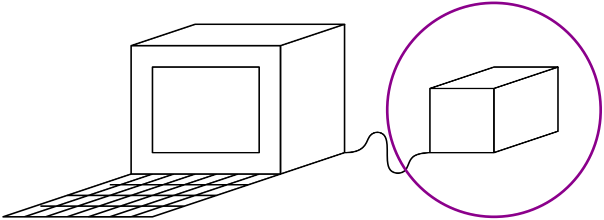


Model of Computation: Co-processor



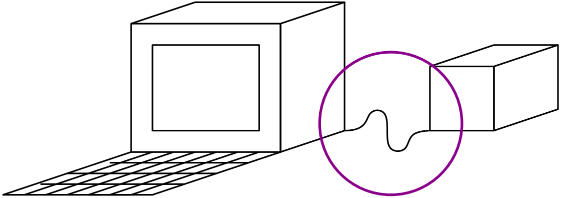
The program lives here

Model of Computation: Co-processor



This only holds the quantum memory

Model of Computation: Co-processor



Series of instructions/feedbacks

The Quantum Memory

A quantum memory

- Contains individually addressable quantum registers (qbits)
- State of n qbits: complex combination of strings of n bits
- E.g. for $n = 3$:

$$\begin{array}{rcl} & -\frac{1}{2} & \cdot\, 0\,0\,0 \\ + & \frac{1}{2} & \cdot\, 0\,0\,1 \\ + & \frac{i}{2} & \cdot\, 1\,1\,0 \\ - & \frac{i}{2} & \cdot\, 1\,1\,1 \end{array}$$

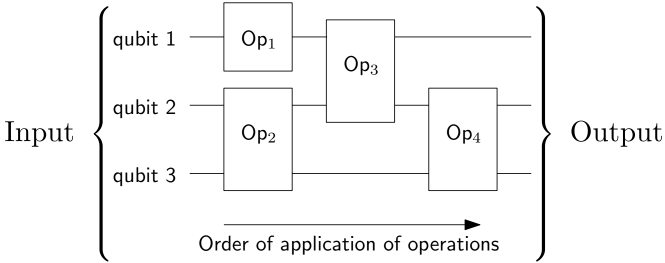
- With a norm condition.

Unlike probabilistic distributions,

all are available at the same time.

Quantum Circuit Model

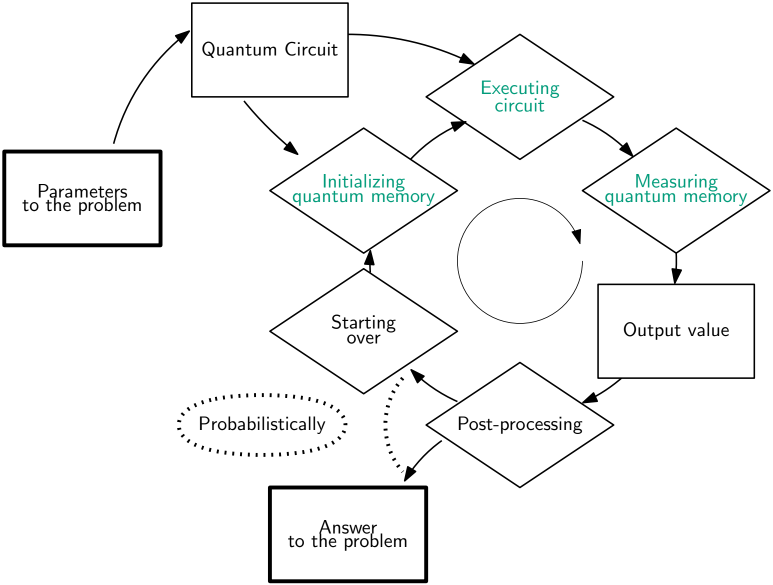
Stream of instructions: a series of elementary gates applied on the quantum memory, that are described by a quantum circuit.



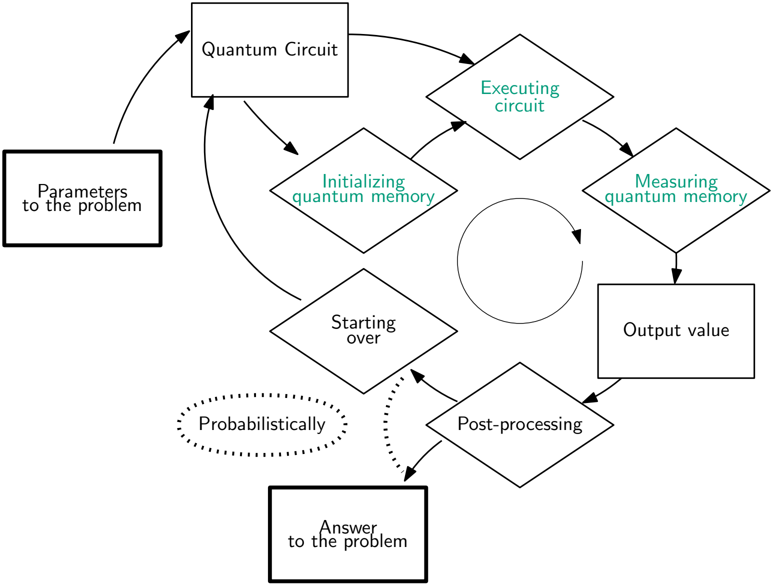
- Each operation is reversible, unitary on the space of states
- Wire $\equiv$ quantum bit $\equiv$ a quantum register
- No "quantum loop", "conditional stop" nor "branching point"

Plan

Structure of (Static) Quantum Algorithms



Structure of (Variational) Quantum Algorithms



Quantum Algorithm, Probabilistic Algorithm

Simple probabilistic algorithm to factor 289884400687823

- Fair draw of a number among $2, 3, 4, 5, \ldots$
- Test: Euclidian division
- Found a factor: success. Otherwise: start over.

Very poor probability of success!

Shor's factorization algorithm

- Probabilistic sampling performed with measurement
- The quantum circuit build a "good" probability distribution.
  $\rightarrow$ boosts factors!

Quantum programming means building a circuit

(In case you're wondering: $315697 \cdot 918236159$)

Internals of Current Quantum Algorithms

The techniques used to described quantum algorithms are diverse.

1. Quantum primitives.
   - Quantum Fourier Transform
     Assuming $\omega = \mathtt{0.xy}$, we want

$$\begin{array}{rl} & (e^{2\pi i\omega})^0 \cdot 00 \\ + & (e^{2\pi i\omega})^1 \cdot 01 \\ + & (e^{2\pi i\omega})^2 \cdot 10 \\ + & (e^{2\pi i\omega})^3 \cdot 11 \end{array} \qquad \longmapsto \qquad 1 \cdot \mathtt{xy}$$

Internals of Current Quantum Algorithms

The techniques used to described quantum algorithms are diverse.

1. Quantum primitives.
   - Phase estimation.
   - Amplitude amplification.
     Qubit 3 in state 1 means good.

$$
\begin{array}{rl} & \alpha_0 \cdot 000 \\ + & \alpha_1 \cdot 011 \\ + & \alpha_2 \cdot 100 \\ + & \alpha_3 \cdot 110 \end{array}
\longmapsto
\begin{array}{rl} & {\scriptstyle \alpha_0} \cdot 000 \\ + & \Large{\alpha_1} \cdot 011 \\ + & {\scriptstyle \alpha_2} \cdot 100 \\ + & {\scriptstyle \alpha_3} \cdot 110 \end{array}
$$

The techniques used to described quantum algorithms are diverse.

1. Quantum primitives.
   - Quantum Fourier Transform
   - Amplitude amplification.
   - Quantum walk.

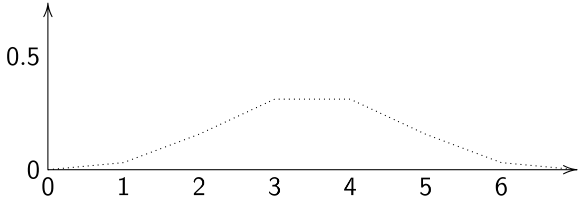
$$
\begin{tikzcd}
0 \arrow[r] & 1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] & 4 \arrow[r] & 5 \arrow[r] & 6 \arrow[r] & 7 \arrow[d] \\
15 \arrow[u] & 14 \arrow[l] & 13 \arrow[l] & 12 \arrow[l] & 11 \arrow[l] & 10 \arrow[l] & 9 \arrow[l] & 8 \arrow[l]
\end{tikzcd}
$$

Internals of Current Quantum Algorithms

The techniques used to described quantum algorithms are diverse.

1. Quantum primitives.
   - Quantum Fourier Transform
   - Amplitude amplification.
   - Quantum walk.

     After 5 steps of a probabilistic walk:

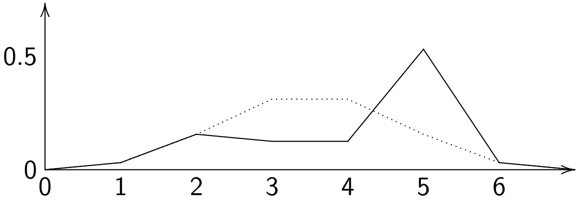


Internals of Current Quantum Algorithms

The techniques used to described quantum algorithms are diverse.

1. Quantum primitives.
   - Quantum Fourier Transform
   - Amplitude amplification.
   - Quantum walk.

     After 5 steps of a quantum walk:



Internals of Current Quantum Algorithms

The techniques used to described quantum algorithms are diverse.

1. Quantum primitives.
   - Quantum Fourier Transform
   - Amplitude amplification
   - Quantum walk
   - Hamiltonian simulation
   - ...

They are given as circuit templates

Internals of Current Quantum Algorithms

The techniques used to described quantum algorithms are diverse.

2. Oracles.
   - Take a classical function $f : \texttt{Bool}^n \to \texttt{Bool}^m$.
   - Construct

$$\begin{array}{rccc} \overline{f}: & \texttt{Bool}^{n+m} & \longrightarrow & \texttt{Bool}^{n+m} \\ & (x, y) & \longmapsto & (x, y \oplus f(x)) \end{array}$$

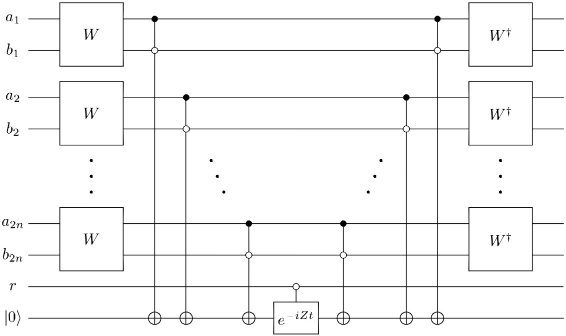
   - Build the unitary U_f acting on $n + m$ qubits computing $\overline{f}$.

Building the circuit depends on how f is given

Internals of Current Quantum Algorithms

The techniques used to described quantum algorithms are diverse.

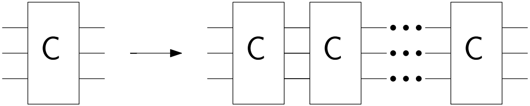
3. Blocks of loosely-defined low-level circuits.



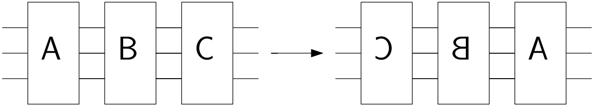
This is not a formal specification!

The techniques used to described quantum algorithms are diverse.

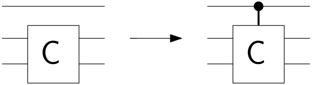
4. High-level operations on circuit:
   - Repetition



   - Inversion



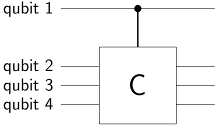
   - Control



Internals of Current Quantum Algorithms

The techniques used to described quantum algorithms are diverse.

4. High-level operations on circuit:
   - Control : conditional action of a circuit



C is applied on qubits 2-4 only when qubit 1 is true:
Suppose that C flips its input bits. Then the above circuit does

$$\begin{array}{rl} \texttt{qbit} & 1\,234 \\ \frac{1}{\sqrt{2}} & \color{blue}{1}\,010 \\ +\frac{1}{\sqrt{2}} & \color{red}{0}\,110 \end{array} \quad \longmapsto \quad \begin{array}{rl} \texttt{qbit} & 1\,234 \\ \frac{1}{\sqrt{2}} & \color{blue}{1\,101} \\ +\frac{1}{\sqrt{2}} & \color{red}{0}\,110 \end{array}$$

This acts as a form of “quantum test”

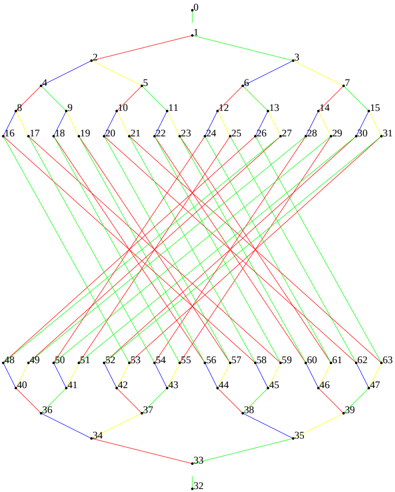
Internals of Current Quantum Algorithms

The techniques used to described quantum algorithms are diverse.

5. Classical processing.
   - Generating the circuit...
   - Computing the input to the circuit.
   - Processing classical feedback in the middle of the computation.
   - Analyzing the final answer (and possibly starting over).

Plan

Case study: BWT algorithm



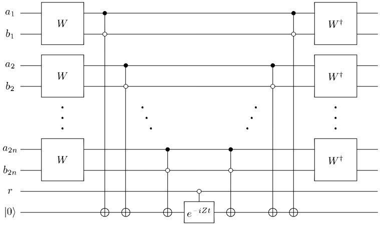
- Start at entrance, look for exit
- Description of the graph:

  I : `Node`
  G : `Color` $\times$ `Node` $\to$ `Maybe Node`
  O : `Node` $\to$ `Bool`

- Random/Quantum walk
- Parameters:
  height of tree ; number of steps.

Case study: BWT algorithm

- Initialization of a register to the input node (using I)
- 10^6 iterations:
  - Diffuse
  - Call oracle for red
  - Diffuse
  - Call oracle for green
  - Diffuse
  - Call oracle for blue
  - Diffuse
  - Call oracle for yellow
- Measure the node we sit on
- Test with O that we reached the output node.



Case study: QLS algorithm

Considering a vector $\vec{b}$ and the system

$$A \cdot \vec{x} = \vec{b},$$

compute the value of $\langle \vec{x} \,|\, \vec{r} \rangle$ for some vector $\vec{r}$.

Practical situation: the matrix A corresponds to the finite-element approximation of the scattering problem:

Case study: QLS algorithm

For more precision: `arXiv:1505.06552`

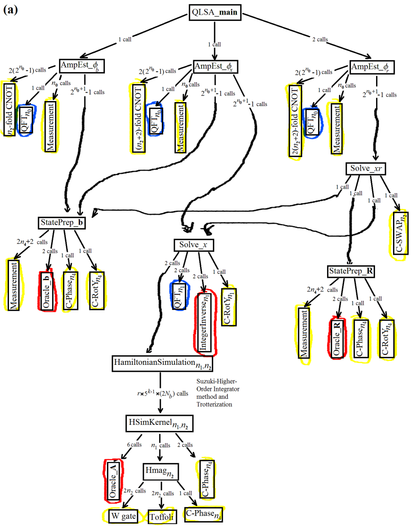
Three oracles:

- for $\vec{r}$ and for $\vec{b}$: input an index, output (the representation of) a complex number
- for A: input two indexes, output also a complex number

It uses many quantum primitives

- Amplitude estimation
- Phase estimation
- Amplitude amplification
- Hamiltonian simulation

Case study: QLS algorithm



- Yellow: Elementary gates.
- Red: Oracles.
- Blue: QFT's.
- Black: Subroutines.
- *Parameters:*
  Dimensions of the space;
  Precision for each of the vectors;
  Allowed error;
  Various parameters for A...
  In total, 19 parameters.

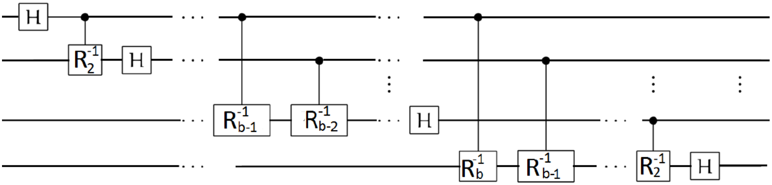
Case study: QLS algorithm

Oracle R is given by the function

```
calcRweights y nx ny lx ly k theta phi =
    let (xc',yc') = edgetoxy y nx ny in
    let xc = (xc'-1.0)*lx - ((fromIntegral nx)-1.0)*lx/2.0 in
    let yc = (yc'-1.0)*ly - ((fromIntegral ny)-1.0)*ly/2.0 in
    let (xg,yg) = itoxy y nx ny in
    if (xg == nx) then
        let i = (mkPolar ly (k*xc*(cos phi)))*(mkPolar 1.0 (k*yc*(sin phi)))*
                ((sinc (k*ly*(sin phi)/2.0)) :+ 0.0) in
        let r = ( cos(phi) :+ k*lx )*((cos (theta - phi))/lx :+ 0.0) in i * r
    else if (xg==2*nx-1) then
        let i = (mkPolar ly (k*xc*cos(phi)))*(mkPolar 1.0 (k*yc*sin(phi)))*
                ((sinc (k*ly*sin(phi)/2.0)) :+ 0.0) in
        let r = ( cos(phi) :+ (- k*lx))*((cos (theta - phi))/lx :+ 0.0) in i * r
    else if ( (yg==1) && (xg<nx) ) then
        let i = (mkPolar lx (k*yc*sin(phi)))*(mkPolar 1.0 (k*xc*cos(phi)))*
                ((sinc (k*lx*(cos phi)/2.0)) :+ 0.0) in
        let r = ( (- sin phi) :+ k*ly )*((cos(theta - phi))/ly :+ 0.0) in i * r
    else if ( (yg==ny) && (xg<nx) ) then
        let i = (mkPolar lx (k*yc*sin(phi)))*(mkPolar 1.0 (k*xc*cos(phi)))*
                ((sinc (k*lx*(cos phi)/2.0)) :+ 0.0) in
        let r = ( (- sin phi) :+ (- k*ly) )*((cos(theta - phi)/ly) :+ 0.0) in i * r
    else 0.0 :+ 0.0
```

Case study: circuit snippets

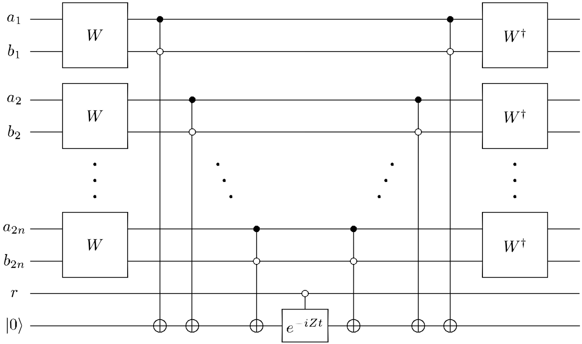
The algorithms create circuits whose sizes and shapes depend on the parameters. E.g. the size of the input register:



(QFT)

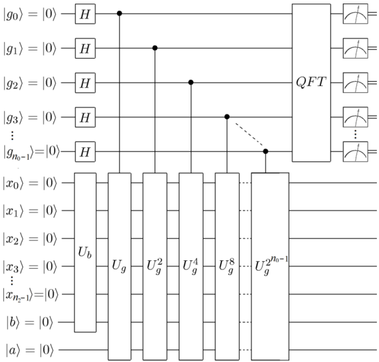
Case study: circuit snippets

The algorithms create circuits whose sizes and shapes depend on the parameters. E.g. the size of the input register:



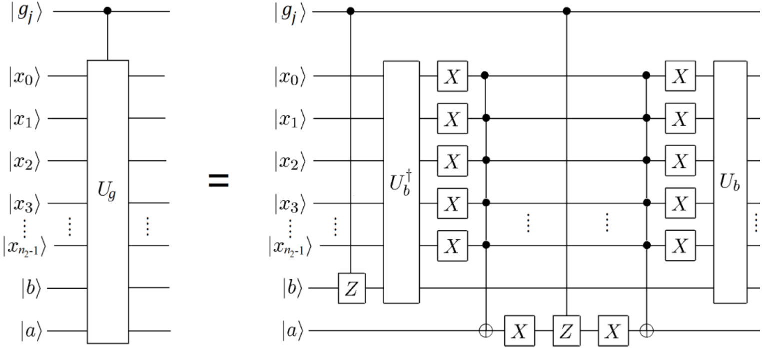
(diffusion step in BWT)

The algorithms create circuits whose sizes and shapes depend on the parameters. E.g. the size of the input register:



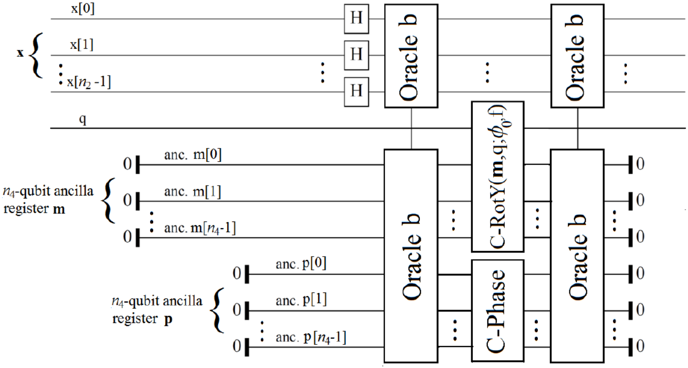
(piece of one subroutine of QLS)

The algorithms create circuits whose sizes and shapes depend on the parameters. E.g. the size of the input register:



(the subroutine U_g)

The algorithms create circuits whose sizes and shapes depend on the parameters. E.g. the size of the input register:



(the subroutine U_b)

Plan

Lessons learned

- Circuit construction
  - Procedural: Instruction-based, one line at a time
  - Declarative: Circuit combinators
    - Inversion
    - Repetition
    - Control
    - Computation/uncomputation
- Circuits as inputs to other circuits
- Regularity with respect to the size of the input
- Distinction parameter / input
- Need for automation for oracle generation

Plan

Programming framework

Two approaches

- Circuit as a record
  - One type `circuit`
  - Qubits $\equiv$ wire numbers
  - Native: vertical/horizontal concatenation, gate addition
- Circuit as a function
  - Qubits $\equiv$ first-order objects
  - Input wires $\equiv$ function input
  - Output wires $\equiv$ function output

Circuits as Records

Simplest model: an object holding all of the circuit structure

- Classical wires
- Quantum wires
- List of gates (or directed acyclic graph)
- This is for instance QisKit/QASM model

In this system

- Static circuit
- No high-level hybrid interaction: sequence
  1. circuit generation
  2. circuit evaluation
  3. measure
  4. classical post-processing
  5. back to (1)

Circuits as Records

Procedural construction (QisKit)

```
q = QuantumRegister(5)
c = ClassicalRegister(1)
circ = QuantumCircuit(q,c)

circ.h(q[0])
for i in range(1,5):
  circ.cx(q[0], q[i])
circ.meas(q[4],c[0])
```

- Static ID For registers
- Wires are numbers
- Gate $\equiv$ instruction
- Classical control: Circuit building
- Explicit "run" of circuit

Combinators: `return a record circuit`

- `circ.control(4)`
- `circ.inverse()`
- `circ.append(other-circuit)`

Circuits as Functions

A function (Quipper)

```
a -> Circ b
```

- Inputs something of type `a`
- Outputs something of type `b`
- As a side-effect, generates a circuit snippet.

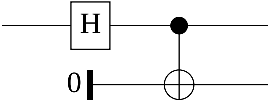
Or

- Inputs a value of type `a`
- Outputs a computation of type `b`

Plan

Circuits as Functions

The circuit



can be typed with

```
Qubit -> Circ (Qubit,Qubit)
```

- Inputs one qubit
- Outputs a pair of qubits
- Spits out some gates when evaluated

The gates are however encapsulated in the function

Circuits as Functions

Representing circuits (Quipper)

Name of circuit
Input: one wire
Indeed a circuit
Two output wires

```
myCircuit :: Qubit -> Circ (Qubit, Qubit)

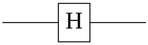
myCircuit q = do
      ...
      ...
      return (x,y)
```

Start a procedural sequence
The name of the input wire
The two output wires

Circuits as Functions

Procedural presentation of circuits:

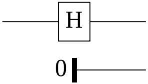
```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r `controlled` q
  return (q,r)
```



Circuits as Functions

Procedural presentation of circuits:

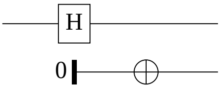
```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r `controlled` q
  return (q,r)
```



Circuits as Functions

Procedural presentation of circuits:

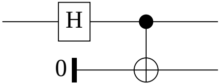
```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r `controlled` q
  return (q,r)
```



Circuits as Functions

Procedural presentation of circuits:

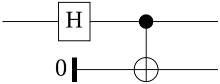
```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r `controlled` q
  return (q,r)
```



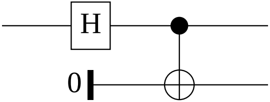
Circuits as Functions

Procedural presentation of circuits:

```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r ‘controlled‘ q
  return (q,r)
```



Circuits as Functions




```
import Quipper

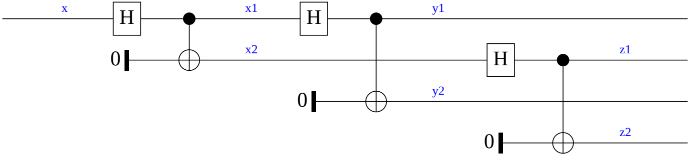
circ ::
   Qubit -> Circ (Qubit,Qubit)
circ x = do
   y <- qinit False
   hadamard_at x
   qnot_at y ‘controlled‘ x
   return (x,y)
```

- Qubits $\equiv$ first-class variable
- Circuit $\equiv$ function
- Wires $\equiv$ inputs and outputs
- Mix classical/quantum

Circuits as Functions

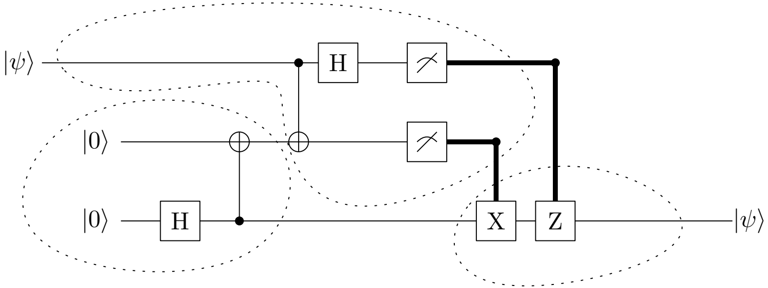
Wires do not have "fixed" location

```
circ2 :: Qubit -> Circ ()
circ2 x = do
   (x1,x2) <- circ x
   (y1,y2) <- circ x1
   (z1,z2) <- circ x2
   return ()
```



- Qubit $\not\equiv$ Wire number
- Circuits as functions: can be applied
- More expressive types

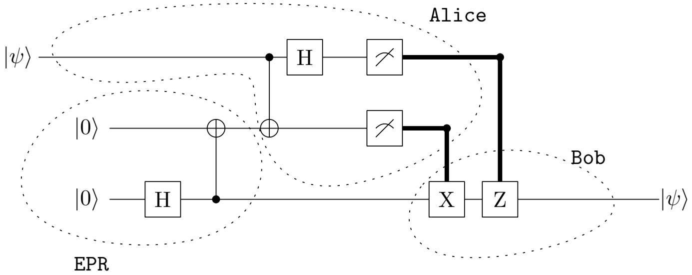
Exercise: Decompose according to the dashed sections



Circuit Combinators: exercise !

What could be the corresponding operations ?

1. `(a -> Circ b) -> (b -> Circ c) -> (a -> Circ c)`
2. `(a -> Circ b) -> (b -> Circ a)`
3. ```
   (a -> Circ b) -> (c -> Circ d)
                 -> ((a,c) -> Circ (b,d))
   ```
4. `(a -> Circ b) -> ((a,Qubit) -> Circ (b,Qubit))`
5. `(a -> Circ b) -> (Qubit -> a -> Circ (b,Qubit))`
6. `(a -> Circ b) -> (Qubit -> a -> Circ b)`
   ```



can be typed as

- `EPR :: Circ (Qubit, Qubit)`
- `Alice :: Qubit -> Qubit -> Circ (Bit,Bit)`
- `Bob :: Qbit -> (Bit,Bit) -> Qubit)`

Composing, we get

```
Circ (Qubit -> Circ (Bit,Bit), (Bit,Bit) -> Circ Qubit)
```

Plan

Families of Circuits

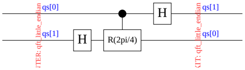
A program

- Inputs classical parameters
- Construct a circuit from these parameters
- Run the circuit

Circuits are parametrized families!

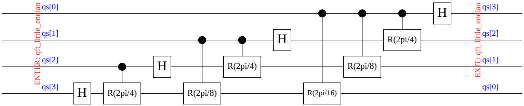
Families of Circuits

Example: QFT



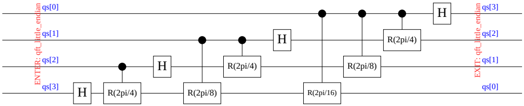
Families of Circuits

Example: QFT



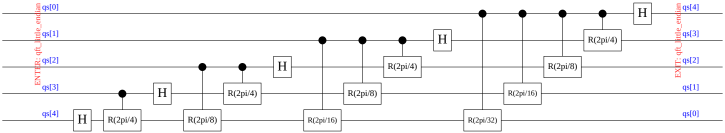
Families of Circuits

Example: QFT



Families of Circuits

Example: QFT



Families of Circuits

With the help of lists:

Name of circuit

Input a **list of wires**

Indeed a circuit

Output a **list of wires**

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

myCircuit qs = do
      ...
      ...
      return ...
```

Start a procedural sequence

The name of the input list

The output list

Families of Circuits

List combinators, e.g.

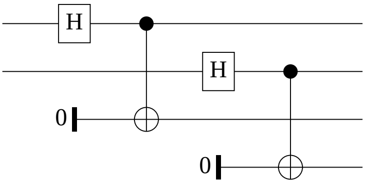
```
mapM ::  (a -> Circ b) -> [a] -> Circ [b]
```

Mixed presentation of circuits:

```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r `controlled` q
  return (q,r)

prog2 :: [Qubit] -> Circ [(Qubit,Qubit)]
prog2 l = mapM prog l
```

List of size 2:



Families of Circuits

List combinators, e.g.

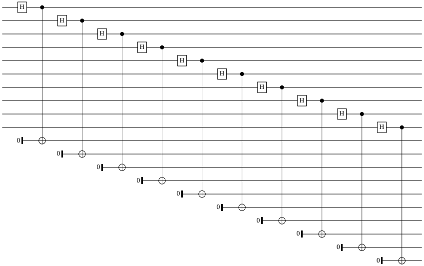
```
mapM ::  (a -> Circ b) -> [a] -> Circ [b]
```

Mixed presentation of circuits:

```
prog :: Qubit -> Circ (Qubit,Qubit)
prog q = do
  hadamard_at q
  r <- qinit False
  qnot_at r ‘controlled‘ q
  return (q,r)

prog2 :: [Qubit] -> Circ [(Qubit,Qubit)]
prog2 l = mapM prog l
```

List of size 10:



Plan

Example: Quipper Code

```
import Quipper

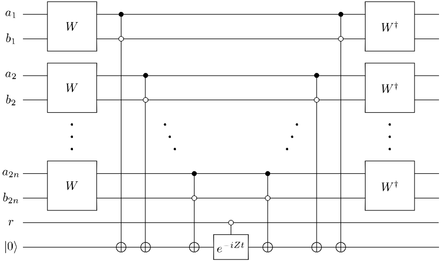
w :: (Qubit,Qubit) -> Circ (Qubit,Qubit)
w = named_gate "W"

toffoli :: Qubit -> (Qubit,Qubit) -> Circ Qubit
toffoli d (x,y) =
  qnot d `controlled` x .==. 1 .&&. y .==. 0

eiz_at :: Qubit -> Qubit -> Circ ()
eiz_at d r =
  named_gate_at "eiZ" d `controlled` r .==. 0

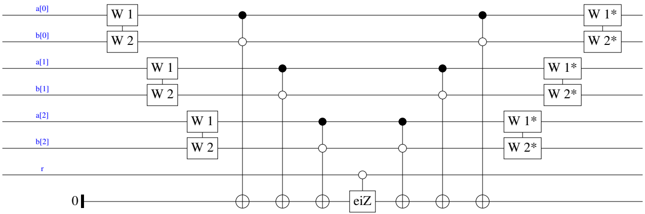
circ :: [(Qubit,Qubit)] -> Qubit -> Circ ()
circ ws r = do
  label (unzip ws,r) (("a","b"),"r")
  d <- qinit 0
  mapM_ w ws
  mapM_ (toffoli d) ws
  eiz_at d r
  mapM_ (toffoli d) (reverse ws)
  mapM_ (reverse_generic w) (reverse ws)
  return ()

main = print_generic EPS circ (replicate 3 (qubit,qubit)) qubit
```



Example: BWT

Result (3 wires):



Example: BWT

Result (30 wires):

Plan

Design Choices: Summary

Requirements for coding Circuits

» Classical structures!

» Hierarchical Representation

» Parametricity

» Non-trivial Combinators

A natural view

» Low Key, First-Order Functions

» Circuit construction seen as a monad

Circuit Construction as a Monad?

Monad: a type constructor `M` equipped with

- `return ::  a -> M a`
- `app ::  M a -> (a -> M b) -> M b`

Example: the list monad

- Type `[a]` : for lists of elements of type a
- `return x = [x]`
- `app [x1, x2, x3] f = (f x1) ++ (f x2) ++ (f x3)`

Circuit Construction as a Monad?

Monad: a type constructor `M` equipped with

- `return ::  a -> M a`
- `app ::  M a -> (a -> M b) -> M b`

Example: state monad `M a = Int -> (a, Int)`

- `return x =` λ `n . (x, n)`
- `app g f =` λ `n . let (y,m) = g n in f y m`

Special combinators

```
get ::  M Int
get = λ n . (n,n)

inc ::  Int -> M ()
inc n = λ m . ((), m+n)
```

do-notation

```
double = do
    n <- get
    inc n
```

Circuit Construction as a Monad?

Monad: a type constructor `M` equipped with

» `return ::  a -> M a`

» `app ::  M a -> (a -> M b) -> M b`

Circuit monad: `M a = GateList -> (a, GateList)`

» `return x =` λ `n . (x, n)`

» `app g f =` λ `n . let (y,m) = g n in f y m`

Special combinators

```
addGate ::  Gate -> Wire -> M ()
addGate g w = λ gs . ((), [(g w) added to gs])
```

```
qinit ::  M Wire
qinit = λ gs . ([fresh wire not in gs], gs)
```

Interaction only performed through these combinators

Quantum PL in the wild

Just to name a few

- Quipper (Academic project)
- Q# (Microsoft)
- Silq (ETH Zurich)

And a wealth of Python's libraries

- Cirq (Google)
- myQLM (Eviden)
- Perceval (Quandela)
- Qiskit (IBM)
- and one for about every single company out there

Plan

Context

An oracle:

- classical description f of the problem
- turned into a reversible circuit:

$$U_f : |x\rangle|y\rangle \mapsto |x\rangle|y + f(x)\rangle$$

- How to build U_f ?
  - Small size: circuit synthesis
  - Arithmetic or other studied functions: Specific (highly optimized) circuits
  - Other cases?

Context

What about an arbitrary program, for example

```
calcRweights y nx ny lx ly k theta phi =
 let (xc',yc') = edgetoxy y nx ny in
 let xc = (xc'-1.0)*lx - ((fromIntegral nx)-1.0)*lx/2.0 in
 let yc = (yc'-1.0)*ly - ((fromIntegral ny)-1.0)*ly/2.0 in
 let (xg,yg) = itoxy y nx ny in
 if (xg == nx) then
     let i = (mkPolar ly (k*xc*(cos phi)))*(mkPolar 1.0 (k*yc*(sin phi)))*
             ((sinc (k*ly*(sin phi)/2.0))+0.0) in
     let r = ( cos(phi)+k*lx )*((cos (theta - phi))/lx+0.0) in i*r
 else if (xg==2*nx-1) then
     let i = (mkPolar ly (k*xc*cos(phi)))*(mkPolar 1.0 (k*yc*sin(phi)))*
             ((sinc (k*ly*sin(phi)/2.0))+0.0) in
     let r = ( cos(phi)+(- k*lx))*((cos (theta - phi))/lx+0.0) in i*r
 else if ( (yg==1) and (xg<nx) ) then
     let i = (mkPolar lx (k*yc*sin(phi)))*(mkPolar 1.0 (k*xc*cos(phi)))*
             ((sinc (k*lx*(cos phi)/2.0))+0.0) in
     let r = ( (- sin phi)+k*ly )*((cos(theta - phi))/ly+0.0) in i*r
 else if ( (yg==ny) and (xg<nx) ) then
     let i = (mkPolar lx (k*yc*sin(phi)))*(mkPolar 1.0 (k*xc*cos(phi)))*
             ((sinc (k*lx*(cos phi)/2.0))+0.0) in
     let r = ( (- sin phi)+(- k*ly) )*((cos(theta - phi)/ly)+0.0) in i*r
 else 0.0+0.0
```

(For QLS there was 10 matlab files of such functions)

Problem Statement

This is the topic of this section. How to:

- in short time
- and automatically
- get efficient,
- scalable,
- yet guaranteed
- reversible implementation
- of a higher-order, classical function,
- parametrically on the input size.

Plan

Basic idea

Landauer's embedding:

- Record all intermediate results.
- With $(x \wedge y) \wedge z$

  $$t \mapsto x \wedge y; \quad u \mapsto t \wedge z; \quad \text{returns } u$$

  while retaining t as "garbage".
- Trace as a partial execution

Basic idea

Example: $x : \texttt{bool} \longmapsto \texttt{let}\ f = \texttt{not}\ \texttt{in}\ (fx)\,\texttt{and}\,(fx) : \texttt{bool}$

Regular execution

- Needs a concrete input, e.g. $x = \texttt{true}$
- Then: rewriting of the term

$$
\begin{array}{rl}
 & \texttt{let}\ f = \texttt{not}\ \texttt{in}\ (f\,\texttt{true})\,\texttt{and}\,(f\,\texttt{true}) \\
\to & (\texttt{not}\,\texttt{true})\,\texttt{and}\,(\texttt{not}\,\texttt{true}) \\
\to & \texttt{false}\,\texttt{and}\,(\texttt{not}\,\texttt{true}) \\
\to & \texttt{false}\,\texttt{and}\,\texttt{false} \\
\to & \texttt{false}
\end{array}
$$

Basic idea

Example: $x : \texttt{bool} \longmapsto \texttt{let}\ f = \texttt{not}\ \texttt{in}\ (fx)\,\texttt{and}\,(fx) : \texttt{bool}$

Trace of a partial execution

- Start with an unknown variable x
- Then: keep the trace of low-level actions to be performed on x

$$
\begin{array}{rll}
 & (\emptyset, & \texttt{let}\ f = \texttt{not}\ \texttt{in}\ (fx)\,\texttt{and}\,(fx)) \\
\to & (\emptyset, & (\texttt{not}\,x)\,\texttt{and}\,(\texttt{not}\,x)) \\
\to & ([y := \texttt{not}\,x] & y\,\texttt{and}\,(\texttt{not}\,x)) \\
\to & ([y \mapsto \texttt{not}\,x; z \mapsto \texttt{not}\,x], & y\,\texttt{and}\,z) \\
\to & ([y \mapsto \texttt{not}\,x; z \mapsto \texttt{not}\,x; t \mapsto y\,\texttt{and}\,z], \quad t) &
\end{array}
$$

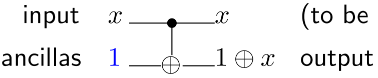
- . . . and make this trace reversible: Landauer's embedding

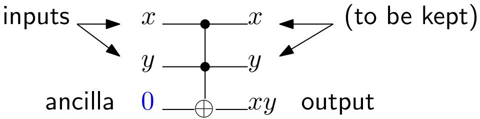
Basic idea

Example: $x : \texttt{bool} \longmapsto \texttt{let}\ f = \texttt{not}\ \texttt{in}\ (fx)\ \texttt{and}\ (fx) : \texttt{bool}$

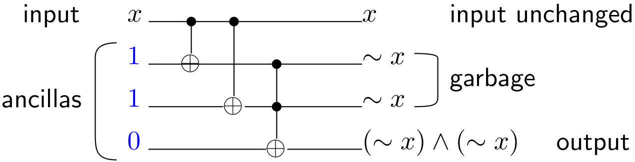
Trace of a partial execution

» `not` becomes a CNOT ; `and` becomes a Toffoli





» And the full trace is



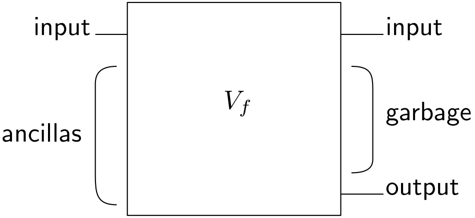
Plan

Composition Procedure

A function f : $\texttt{bool} \longrightarrow \texttt{bool}$
is turned into a map

$$V_f : \texttt{bool} \longrightarrow \mathrm{circuit}(\texttt{bool})$$

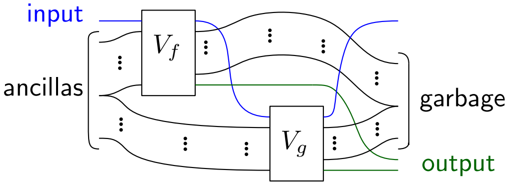
(Note: omit garbage in type)



Composition Procedure

A function $\langle f, g\rangle : \texttt{bool} \longrightarrow (\texttt{bool} \times \texttt{bool})$
is turned into a map

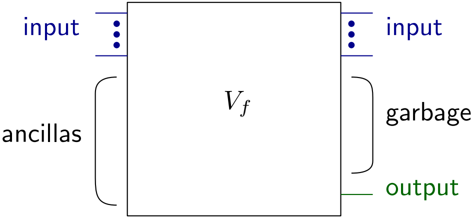
$$V_{\langle f,g\rangle} : \texttt{bool} \longrightarrow \mathrm{circuit}(\texttt{bool}\times\texttt{bool})$$



Composition Procedure

A function $f : (\texttt{bool list}) \longrightarrow \texttt{bool}$
is turned into a map

$$V_f : (\texttt{bool list}) \longrightarrow \mathrm{circuit}(\texttt{bool})$$

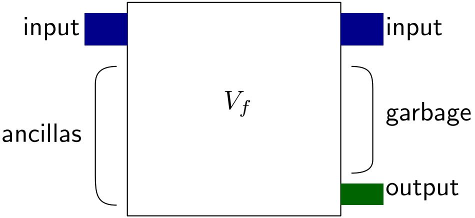


(Parametric circuit !)

Composition Procedure

A function $f : A \longrightarrow B$
is turned into a map

$$f : A \longrightarrow \mathrm{circuit}(\text{“}B\text{”})$$

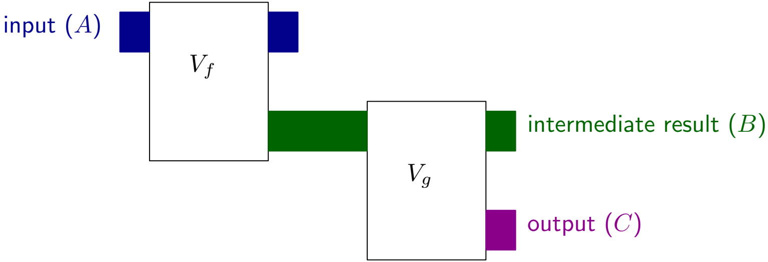


Composition Procedure

Two function $f : A \longrightarrow B$ and $g : B \to C$
are turned into maps

$$V_f : A \longrightarrow \text{circuit}(``B")$$
$$V_g : B \longrightarrow \text{circuit}(``C")$$

Composition $g \circ f : A \longrightarrow C$ is turned into $A \longrightarrow \text{circuit}(``C")$



Composition Procedure

Example Try out

$$(x, y) \longmapsto \neg(\neg x) \wedge (\neg y)$$

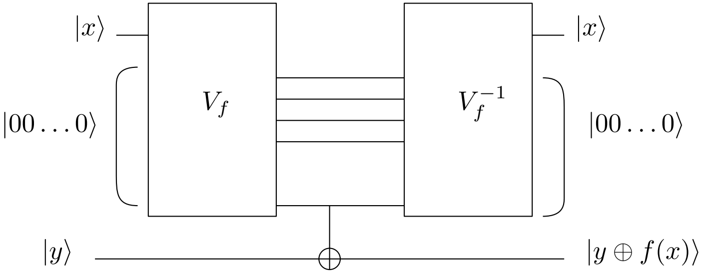
Example Try out

$$x \longmapsto x^8$$

Assume x is a natural number modulo 2^N written as a bitstring of size N, and assume that we already have a very optimized V for the multiplication.

Oracle from V

We construct U as



This scheme is known as compute-uncompute.
It has been implemented in Quipper.

Plan

Example: Adder

```
foldl :: (A → B → A) → A → [B] → A
foldl f a l = let rec g z l' = match (split l') with
                         nil   ↦ z
                       | ⟨h,t⟩ ↦ g (f z h) t
              in g a l

bit_adder : bit → bit → bit → (bit × bit)
bit_adder carry x y =
      let majority a b c = if (xor a b) then c else a in
      let z = xor (xor carry x) y in
      let carry' = majority carry x y in ⟨carry', z⟩

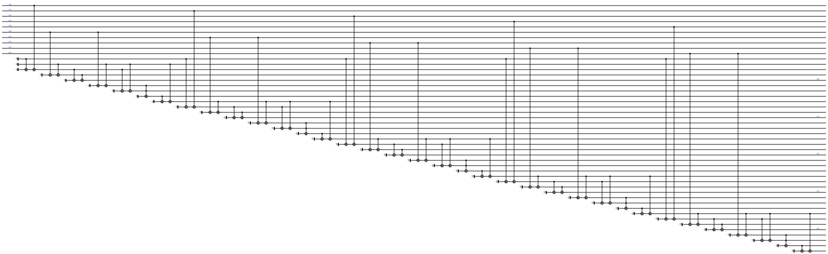
adder_aux : (bit × [bit]) → (bit × bit) → (bit × [bit])
adder_aux ⟨w, cs⟩ ⟨a, b⟩ = let ⟨w', c'⟩ = bit_adder w a b in ⟨w', c'::cs⟩

adder : [bit] × [bit] → [bit]
adder x y = snd (foldl adder_aux ⟨False, nil⟩ (zip y x))
```

`adder` is lifted to `[bit]` $\times$ `[bit]` $\to$ `circuit([bit])`.

Example: Adder

For $n = 5$, no optimization:



Size of circuit is proportional to number of low-level bit-operations in all execution paths of `adder`.

Example: Adder

n is the integer-size in bits.

paper	**ancillae**	**size**
VBE (1995)	n	$\sim 8n$
Cuccaro, Drapper & al. (2005)	0	$\sim 7n$
Drapper, Kutin & al. (2008)	$\sim 2n$	$\sim 10n$ (in place)
Drapper, Kutin & al. (2008)	$\sim n$	$\sim 5n$ (not in place)

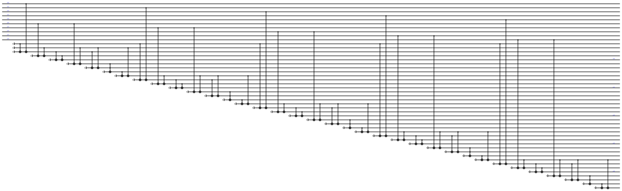
How do we scale against these ?

Example: Adders

If $n = 5$:

paper	**ancillae**	**size**
VBE (1995)	5	~ 40
Cuccaro, Drapper & al. (2005)	0	~ 35
Drapper, Kutin & al. (2008)	~ 10	~ 50 (in place)
Drapper, Kutin & al. (2008)	~ 5	~ 50 (not in place)

Automatically generated (no optimization):

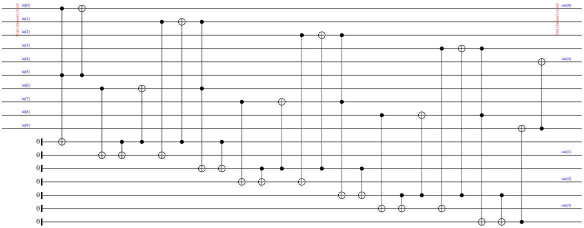


Example: Adders

If $n = 5$:

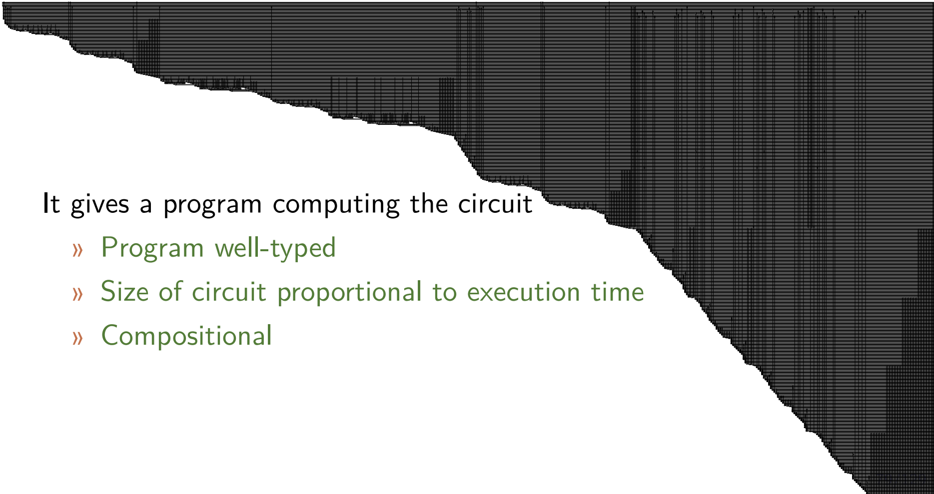
paper	ancillae	size
VBE (1995)	5	~ 40
Cuccaro, Drapper & al. (2005)	0	~ 35
Drapper, Kutin & al. (2008)	~ 10	~ 50 (in place)
Drapper, Kutin & al. (2008)	~ 5	~ 50 (not in place)

With a bit of (automated) optimization:



Example: The vector b

Hand-made circuits for: adders, multipliers, comparison, square root.
How about the b vector of the QLS algorithm ($Ax = b$) ?

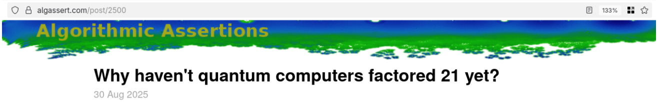


It gives a program computing the circuit

- Program well-typed
- Size of circuit proportional to execution time
- Compositional

Oracle Synthesis Nowadays

A lot of progress! But not quite enough to get there
See for example Gidney's blog:



For Shor's factoring algoritm, aiming at 21:

More broadly

» Realm of FTQC
» Large actors enters the field: Google, Microsoft, IBM, AWS. . .
» . . . and small actors such as Alice&Bob or PsiQuantum

Plan

Plan

Goal

Formalizing

- Functional programming
- Higher-order combinators
- Capable of manipulating quantum information
- And quantum circuits

The first two items

- Realm of lambda-calculus

Lambda-Calculus

- Formal system from Alonzo Church, $\sim$1930
- Concept of Function and Application
  - "Every term is a function!"
  - Core of functional programming
  - For example: Haskell, OCaml, F#, Lisp, Erlang, *etc.*
- Very simple grammar:
  - Variables $x_1, x_2, x_3, \ldots$
  - Application (binary, infix)
  - Abstraction: $\lambda x.t$ (where t is a term)
- $\lambda x.t$: the function "$x \mapsto t$"

Lambda-Calculus

- Extension of a first-order system
  - Can be extended to other first-order symbols
  - Also to other second-order constructions (like μ-calculus...)
  - But universal
- Notation
  - $\lambda x.t_1\, t_2\, t_3 = \lambda x.((t_1\, t_2)\, t_3)$
  - $\lambda xy.t = \lambda x.\lambda y.t$

Lambda-Calculus

- Notion of bound and free variables
- In $\lambda y.x(\lambda z.z)$:
  - x is free
  - y, z are bound
- Each bound variable is attached to a λ
  - $\lambda z\ .\ x\ \lambda x\ .\ x\ (\ \lambda x\ .\ x\ z\ )$
- The name of bound variables does not matter
  - $\lambda x.x = \lambda y.y$
  - $\lambda xy.x\,(y\,z) = \lambda ab.a\,(b\,z)$
  - Careful! $\lambda x.y \neq \lambda y.y$

Rewriting Rules

- β-reduction: $(\lambda x.t)u \longrightarrow_\beta t[x := u]$
  (**!** only the x bound by the corresponding λ are replaced)

- A rule that can be added:
  η-reduction: $(\lambda x.tx) \longrightarrow_\eta t$
  (**!** when x is not free in t)

- Congruence:
  Reduction can occur within a term
  If $M \longrightarrow M'$, then $MN \longrightarrow M'N$
  (Context-free rules)

Example

What are the behaviors of

- $\Omega = (\lambda x.xx)(\lambda x.xx)$?
- $ZZ\,V$ when $Z = \lambda zx.x(zzx)$? (Turing's fixed point combinator)

Church numerals are defined as

- $\overline{0} \triangleq \lambda xy.y$
- $\overline{1} \triangleq \lambda xy.xy$
- $\overline{2} \triangleq \lambda xy.x(xy)$
- $\overline{3} \triangleq \lambda xy.x(x(xy))$

When fed with $\overline{m}$ and $\overline{n}$, what are the behaviors of

- $M = \lambda mn.\lambda xy.mx(nxy)$?
- $N = \lambda mn.m(Mn)(\lambda xy.y)$?

Pure lambda calculus is Turing complete

Link with Functional Programming

$(\lambda x.t)u$ can be interpreted as let $x = u$ in t

Example with

```
let a = 1 + 2 in
let b = 5 * a * a in
let f = λx . a * x * x in f b
```

(Here we assume that we have integers available somehow)

Evaluation Strategy

- Imagine that "`tic`" evaluates to 0 while emitting... a tic.
- Consider the term:

$$(\lambda x.xx)((\lambda yz.z)\,\texttt{tic})$$

- How many tics does the term emit?

Two standard strategies

- No rewriting under λ's
- Call by name: "As far left as possible" / "As early as possible"
  This is called lazy evaluation
  $\rightarrow$ Evaluation used in Haskell, for example
- Call by value: "As far right as possible"
  Evaluation starts with the arguments
  $\rightarrow$ The standard evaluation: OCaml, F#, etc.

Call-by-Value

```
let a = 1 + 2 in
let b = 5 * a * a in
let f = λx . a * x * x in f b
```

corresponds to the lambda term:

$$(\lambda a.(\lambda b.(\lambda f.fb)(\lambda x.a * x * x))(5 * a * a))(1 + 2)$$

Call by value evaluation:

$$\begin{aligned}
&\to (\lambda a.(\lambda b.(\lambda f.fb)(\lambda x.a * x * x))(5 * a * a))3 \\
&\to (\lambda b.(\lambda f.fb)(\lambda x.3 * x * x))(5 * 3 * 3) \\
&\to (\lambda b.(\lambda f.fb)(\lambda x.3 * x * x))45 \\
&\to (\lambda f.f\,45)(\lambda x.3 * x * x) \\
&\to (\lambda x.3 * x * x)45 \\
&\to 3 * 45 * 45 \\
&\to 6075
\end{aligned}$$

No reduction under lambdas

Call-by-Value

```
let a = 1 + 2 in
let b = 5 * a * a in
let f = λx . a * x * x in f b
```

corresponds to the code transformation

```
let a = 3 in
let b = 5 * a * a in
let f = λx . a * x * x in f b
```

which rewrites to

```
let b = 5 * 3 * 3 in
let f = λx . 3 * x * x in f b
```

which rewrites to

```
let b = 45 in
let f = λx . 3 * x * x in f b
```

...

Functional Purity

```
let a = 1 + 2 in
let b = 5 * a * a in
let f = λx . a * x * x in
let a = 0 in f b
```

The second a has nothing to do with the first one

- Immutable variables
- Notion of purely functional language
- The environment does not affect the behavior of a function

Functional Purity

```
let f = λx . λy . x * y in (f (1 + 2)) (3 + 4)
```

or

```
let f = λy . λx . x * y in (f (3 + 4)) (1 + 2)
```

The order of argument evaluation does not matter

- First a then b, or the opposite
- Again linked to functional purity

Simple Type System

Similar to Emmanuel's minimal logic

$$A, B \quad ::= \quad \texttt{nat} \mid A \to B$$

with an opaque type $\texttt{nat}$.

Intuition

- $\lambda x.M$ is typed with $A \to B$ (if well-typed)
- 2 is typed with $\texttt{nat}$
- $+$ is typed with $\texttt{nat} \to \texttt{nat} \to \texttt{nat}$ (modulo infix notation)

Simple Type System

Typing context

$$x_1 : A_1, x_2 : A_2, \ldots x_n : A_n \vdash M : B$$

Typing rules

$$\frac{\Delta, x : A \vdash M : B}{\Delta \vdash \lambda x.M : A \to B} \qquad \frac{\Delta \vdash M : A \to B \quad \Delta \vdash N : A}{\Delta \vdash MN : B}$$

$$\frac{}{\Delta, x : A \vdash x : A}$$

$$\frac{}{+ : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}} \qquad \frac{}{2 : \mathtt{nat}} \qquad \text{(one for each } n\text{)}$$

Simple Type System

Typing context

$$A_1, \quad A_2, \ldots \quad A_n \vdash \quad B$$

Typing rules

$$\frac{\Delta, \quad A \vdash \quad B}{\Delta \vdash \quad A \to B} \qquad \frac{\Delta \vdash \quad A \to B \quad \Delta \vdash \quad A}{\Delta \vdash \quad B}$$

$$\frac{}{\Delta, \quad A \vdash \quad A}$$

Curry-Howard correspondence: well-typed terms are proofs

Extensions

Example: Pairing, Boolean values:

$$A, B \quad ::= \quad \mathtt{nat} \mid A \to B \mid A \times B \mid \mathtt{bit}$$

Typing rules with new term constructs

$$\frac{\Delta, x : A \vdash M : B}{\Delta \vdash \lambda x.M : A \to B} \qquad \frac{\Delta \vdash M : A \to B \quad \Delta \vdash N : A}{\Delta \vdash MN : B}$$

$$\frac{\Delta \vdash M : A \quad \Delta \vdash N : B}{\Delta \vdash \langle M, N \rangle : A \times B} \qquad \frac{\Delta \vdash M : A_1 \times A_2}{\Delta \vdash \pi_i M : A_i}$$

$$\frac{\Delta \vdash P : \mathtt{bit} \quad \Delta \vdash M, N : C}{\Delta \vdash \mathtt{if}\ P\ \mathtt{then}\ M\ \mathtt{else}\ N : C} \qquad \frac{}{\Delta \vdash \mathtt{true}, \mathtt{false} : \mathtt{bit}}$$

Safety Properties

- Subject reduction: type is preserves by reduction
- Progress: well-typed terms either reduce or reached a value

Plan

One problem: Entanglement

Let us add

- an opaque type `qbit`
- constants $|\phi\rangle$ for all possible states

We can do

$$\lambda f.(f\,|0\rangle)\,|1\rangle : (\texttt{qbit} \to \texttt{qbit} \to A) \to A$$

but what if the two states are entangled:

$$\lambda f.(f q_1) q_2 : (\texttt{qbit} \to \texttt{qbit} \to A) \to A$$

where q_1, q_2 is in state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$?

Quantum Lambda-Calculus

Terms

- Pairing constructs and fixpoints
- Boolean `true` and `false`, if-then-else
- Constant, opaque terms: `qinit`, `measure`, `H`, `CNOT`, ...
- Quantum states not in the language
  $\rightarrow$ included as pointers

Operational semantics

- Abstract machine encapsulating the quantum memory:

$$\Big(\underbrace{\tfrac{1}{\sqrt{2}}(|00\rangle + |11\rangle)}_{\text{state vector}}, \quad \underbrace{|xy\rangle}_{\text{“linking function”}}, \quad \underbrace{\lambda f.f\langle x, y\rangle}_{\text{lambda-term}} \Big)$$

- Call-by-value evaluation strategy

  (Reduction strategy linked to the type system!)

- Quantum operations through the evaluation strategy

$$[\alpha\,|0\rangle + \beta\,|1\rangle\,,|x\rangle\,,\texttt{let y = qinit false in CNOT}\ \langle\texttt{x, y}\rangle]$$

reduces to

$$[\alpha\,|00\rangle + \beta\,|11\rangle\,,|xy\rangle\,,\langle x, y\rangle]$$

Another Problem: Non-Duplicability

Consider the following

$$\left[\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), |x\rangle\,, \langle \mathtt{meas}\, x, \mathtt{had}\, x\rangle\right]$$

In a purely functional world, the order should not matter!

Notion of linear type system

- Quantum data is non-duplicable
- Type system based on linear logic

Plan

Quantum Memory

Mathematical Structure

» Quantum register $\equiv$ (finite) Hilbert space

» Juxtaposition $\equiv$ Kronecker (tensor) product

» Reading $\equiv$ measure $\equiv$ getting a bit, probabilistic

» Quantum information is non-duplicable

Type Structure

» Based on (Intuitionistic) Multiplicative Linear Logic

$$A, B \quad ::= \quad \mathtt{qbit} \mid \mathtt{bit} \mid A \otimes B$$

» Entanglement:

$$[\tfrac{1}{\sqrt{2}}(|00\rangle + |11\rangle), |xy\rangle, \langle x, y\rangle] \quad : \quad \mathtt{qbit} \otimes \mathtt{qbit}$$

Quantum Memory

Mathematical Structure

- Quantum register $\equiv$ (finite) Hilbert space
- Juxtaposition $\equiv$ Kronecker (tensor) product
- Reading $\equiv$ measure $\equiv$ getting a bit, probabilistic
- Quantum information is non-duplicable

Type Structure

- Based on (Intuitionistic) Multiplicative Linear Logic

$$A, B \quad ::= \quad \texttt{qbit} \mid \texttt{bit} \mid A \otimes B \mid \underbrace{A \multimap B}_{\text{Type of linear functions}}$$

- Entanglement:

$$[\tfrac{1}{\sqrt{2}}(|00\rangle + |11\rangle), |xy\rangle, \langle x, y\rangle] \quad : \quad \texttt{qbit} \otimes \texttt{qbit}$$

Linear Type System

Core Typing Rules

$$\frac{\Delta, x : A \vdash M : B}{\Delta \vdash \lambda x.M : A \multimap B} \qquad \frac{\Delta \vdash M : A \multimap B \quad \Gamma \vdash N : A}{\Delta, \Gamma \vdash MN : B}$$

$$\frac{\Delta \vdash M : A \quad \Gamma \vdash N : B}{\Delta, \Gamma \vdash \langle M, N \rangle : A \otimes B} \quad \frac{\Delta, x : A, y : B \vdash N : C \quad \Gamma \vdash M : A \otimes B}{\Delta, \Gamma \vdash \texttt{let}\ \langle \texttt{x}, \texttt{y} \rangle\ \texttt{=}\ M\ \texttt{in}\ N : C}$$

$$\frac{}{x : A \vdash x : A}$$

- Non-duplicability
- $\lambda x.\langle x, x \rangle$ is not typable

Quantum Circuit Model

In this model

- Circuit $\equiv$ pure function from input to output

1-qbit unitary	$\mathtt{qbit} \multimap \mathtt{qbit}$
2-qbit unitary	$\mathtt{qbit} \otimes \mathtt{qbit} \multimap \mathtt{qbit} \otimes \mathtt{qbit}$
measurement	$\mathtt{qbit} \multimap \mathtt{bit}$
initialization	$1 \multimap \mathtt{qbit}$
discard	$\mathtt{qbit} \multimap 1$

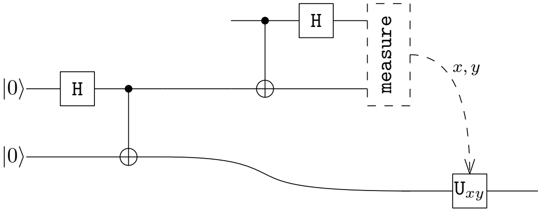
- Vertical composition $\equiv$ tensoring
- Horizontal composition $\equiv$ function composition
- Abstracts away the notion of register

Limitation

- Difficult to implement combinators

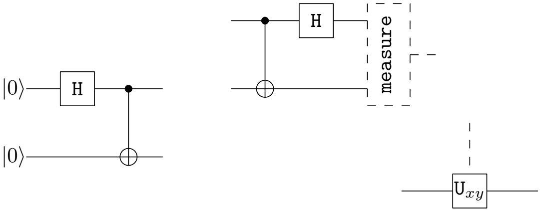
Entanglement of Functions

Example of higher-order entanglement: Teleportation



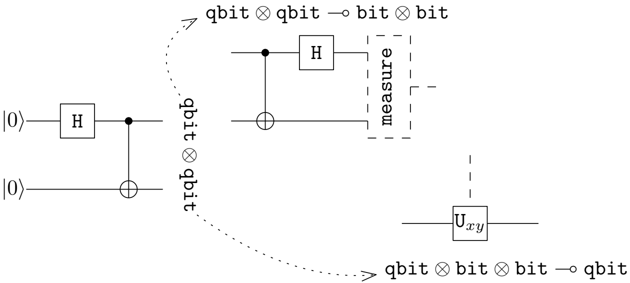
Entanglement of Functions

Example of higher-order entanglement: Teleportation



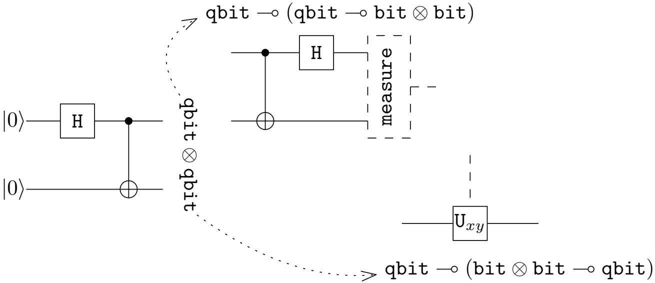
Entanglement of Functions

Example of higher-order entanglement: Teleportation



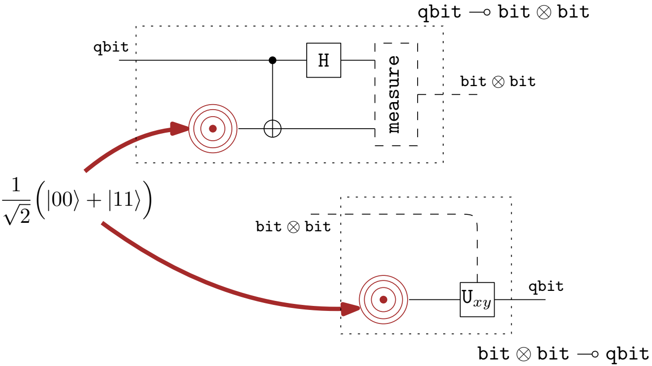
Entanglement of Functions

Example of higher-order entanglement: Teleportation



Entanglement of Functions

Example of higher-order entanglement: Teleportation



A pair of two entangled functions

$$(\mathtt{qbit} \multimap \mathtt{bit} \otimes \mathtt{bit}) \otimes (\mathtt{bit} \otimes \mathtt{bit} \multimap \mathtt{qbit})$$

inverses of each other.

Typing Duplication

A new type construct

$$A, B \ ::= \ \texttt{qbit} \mid \texttt{bit} \mid 1 \mid A \otimes B \mid A \multimap B \mid\ !A$$

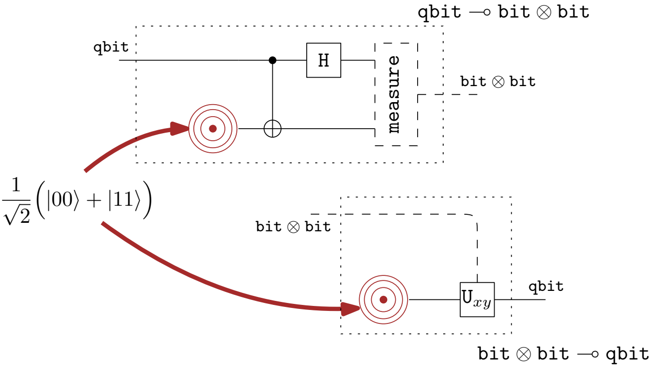
- Based on linear logic
- Non-duplicable functions with $A \multimap B$
- Duplicable functions with $!(A \multimap B)$
- Quantum operations are duplicable
  $\rightarrow$ e.g. $\texttt{measure} : !(\texttt{qbit} \multimap \texttt{bit})$

Non-trivial mix

- Classical and quantum data, probabilistic setting
- Entanglement at higher-order

Entanglement of Functions

Example of higher-order entanglement: Teleportation



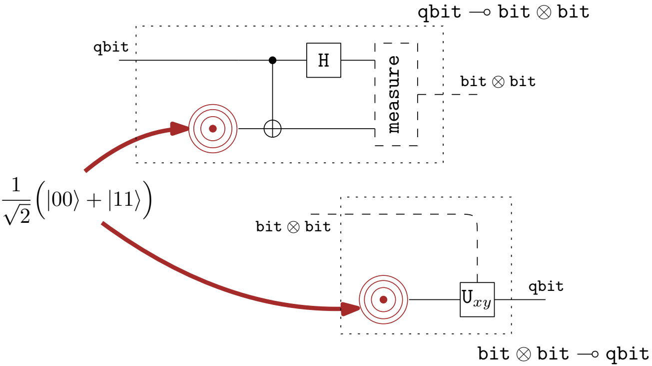
A pair of two entangled non-duplicable functions

$$(\texttt{qbit} \multimap \texttt{bit} \otimes \texttt{bit}) \otimes (\texttt{bit} \otimes \texttt{bit} \multimap \texttt{qbit})$$

inverses of each other.

Entanglement of Functions

Example of higher-order entanglement: Teleportation



A duplicable procedure generating non-duplicable functions

$$!(1 \multimap (\mathtt{qbit} \multimap \mathtt{bit} \otimes \mathtt{bit}) \otimes (\mathtt{bit} \otimes \mathtt{bit} \multimap \mathtt{qbit}))$$

inverses of each other.

Typing Duplication

Core typing rules

$$\frac{!\Delta \vdash V : A}{!\Delta \vdash V : !A}\ (P) \qquad \frac{\Delta \vdash M : !A}{\Delta \vdash M : A}\ (D)$$

$$\frac{\Delta, x :!A, y : !A \vdash M : B}{\Delta, x : !A \vdash M[y := x] : B}\ (C)$$

- Only values can be duplicated

(Call-by-value!)

Examples

- $\vdash$ `had (qinit true)` : !`qbit` (What is wrong?)
- $\vdash \lambda x.\langle x, x\rangle : !A \multimap !A \otimes !A$ (Why?)

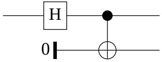
Typing Duplication

Type these terms!

» $\texttt{true}$

» $\lambda x.x$

» $\lambda x.(\mathsf{let}\, z = \mathtt{H}\, x \,\mathsf{in}\, \mathtt{CNOT}\,\langle z, \mathtt{qinit\ false}\rangle)$



» $\mathtt{H}\,(\mathtt{qinit\ false})$

» $\langle\, \mathtt{H}\,(\mathtt{qinit\ false}),\ \lambda x.x\,\rangle$

» $\mathsf{let}\, y = \mathtt{H}\,(\mathtt{qinit\ false})\,\mathsf{in}\, \lambda f.f\, y$

Distinction between

» Procedure for generating a qbit: duplicable

» End result of the procedure: qbit value, non-duplicable

Quantum Lambda-Calculus

Bottom line

- Classical data handled natively
- Quantum data handled through pointers and instructions
- Mix of duplicable and non-duplicable data, with higher-order

Properties

- Type system imposed as axioms
- Safety properties derived "by hand"

(Could have used a realizability approach!)

Limitations

- Gates handled individually
- Far from what is done in quantum algorithm
- Need to consider an extended circuit model

Plan

The problem

Naïve approach for typing combinators:

- Repetition : $\mathbb{N} \multimap (A \multimap A) \multimap (A \multimap A)$ (OK)
- Inversion : $(A \multimap B) \multimap (B \multimap A)$ (WRONG)
- Control : $(A \multimap B) \multimap (\mathtt{qbit} \otimes A \multimap \mathtt{qbit} \otimes B)$ (WRONG)

Does not work

- Would require "reading" the gates.
- But gates can only be sent to the QRAM

Circuit Description Languages

Extending the quantum λ-calculus with

- A new opaque type for circuits: $\mathtt{Circ}(A, B)$
- Box and unbox constructions

$$(A \multimap B) \underset{\mathtt{unbox}}{\overset{\mathtt{box}}{\rightleftarrows}} \mathtt{Circ}(A, B)$$

  - Box: instantiate a new circuit
  - Unbox: evaluate a circuit

- A list of fixed, opaque circuits combinators such as

$$\begin{array}{lcl} \mathtt{ctl} & : & \mathtt{Circ}(A, B) \multimap \mathtt{Circ}(\mathtt{qbit} \otimes A, \mathtt{qbit} \otimes B) \\ \mathtt{rev} & : & \mathtt{Circ}(A, B) \multimap \mathtt{Circ}(B, A) \end{array}$$

- Nice arrow-like, categorical semantics

Formalization of Quipper

- Proto-Quipper
- Notion of circuit-description language

Circuit Description Languages

Possible extensions

- Inductive types (such as lists)
- First-order quantifiers
  $\rightarrow$ limited to "classical types"

Dependent Proto-Quipper

- Type $[A]_n$ for lists of length n made of elements of type A
- In general, $B(n)$ is a type parameterized by n
- $f : \forall n : A \cdot B(n)$: function A to B, with $f(n)$ of type $B(n)$

Examples

- $\forall n : \mathbb{N} \cdot \mathtt{Circ}([\mathtt{qbit}]_n, [\mathtt{qbit}]_n)$
- $\forall m, n : \mathbb{N} \cdot [[\mathtt{qbit}]_m]_n \multimap [\mathtt{qbit}]_{mn}$

Extended Quantum Circuit Model

Summary

- Circuits are now first-order citizens
- Close to what is done for “real algorithms”
- Suitable for formalization and extensions
  [Lindenhovius,Mislove,Zamdzhiev, 2018] [Fu, Selinger et al, 2020,2022,2024] [Lee, V et al, 2021] [Colledan, Dal Lago, 2025]

Limitations

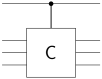
- Circuits are built from opaque boxes
- The only control is classical

Plan

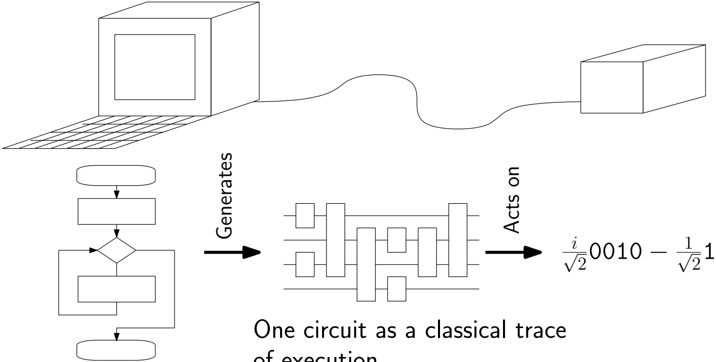
Plan

Notion of Control Flow

Control flow in quantum computation
has two meanings

» The control of a gate 

» The (classical) control-flow of the program:



$\frac{i}{\sqrt{2}}0010 - \frac{1}{\sqrt{2}}1101$

One circuit as a classical trace of execution

Quantum SWITCH

Are circuits “complete” ?

- Canonical model for “usual” quantum algorithms
- But a circuit is causally ordered

The quantum SWITCH

- One copy of $-\boxed{U}-$ and $-\boxed{V}-$
- Want a device with two wires x and y doing

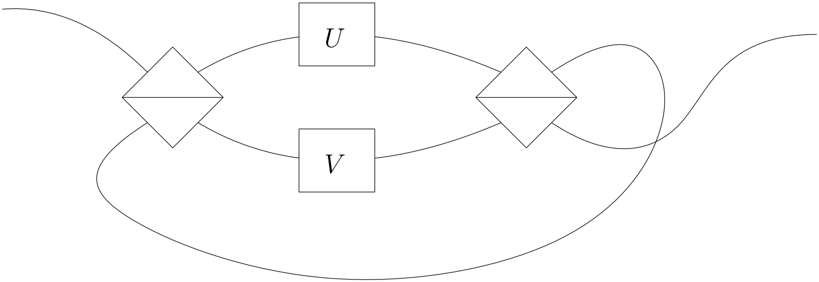
$-\boxed{U}-\boxed{V}-$ on y if x is 0

$-\boxed{V}-\boxed{U}-$ on y if x is 1

Quantum SWITCH

Implementation with quantum photonics

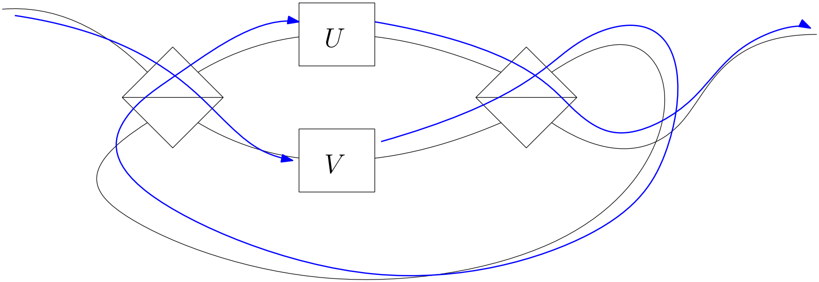
- Single photon
- Control qubit as polarization



Quantum SWITCH

Implementation with quantum photonics

- Single photon
- Control qubit as polarization

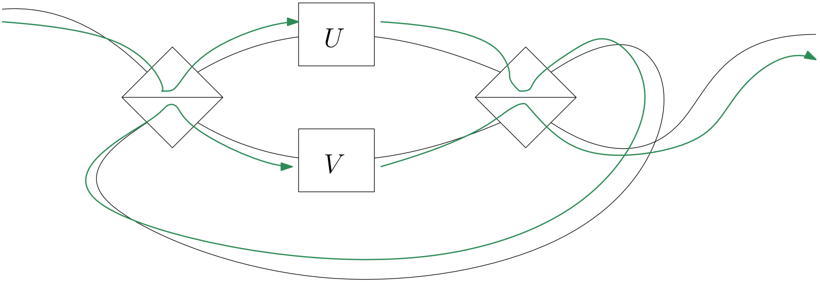


Vertical polarization: goes through and yields V then U.

Quantum SWITCH

Implementation with quantum photonics

- Single photon
- Control qubit as polarization



Horizontal polarization: bounce and yields U then V.

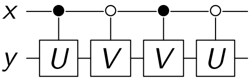
Quantum SWITCH

Quantum Circuit for the quantum SWITCH

- When U and V are duplicable:

$$!\texttt{Circ}(\texttt{qbit},\texttt{qbit}) \multimap !\texttt{Circ}(\texttt{qbit},\texttt{qbit}) \multimap \texttt{Circ}(\texttt{qbit},\texttt{qbit})$$

- Realized with



- What if they are not duplicable?

$$\texttt{Circ}(\texttt{qbit},\texttt{qbit}) \multimap \texttt{Circ}(\texttt{qbit},\texttt{qbit}) \multimap \texttt{Circ}(\texttt{qbit},\texttt{qbit})$$

- Doable with photonics

  [Chiribella,D'Ariano,Perinotti,V] [Oreshkov,Costa,Brukner]

- But no satisfactory notion of quantum circuit

  [Chiribella,D'Ariano,Perinotti,V,2013]

The problem

Building circuit combinators

» Quantum SWITCH as a primitive circuit combinator

» But not really satisfactory!

» How to program circuit combinators?

The circuit construction is CLASSICAL

» Instantiated on one particular set of qubits

» Applied regardless of the state of the memory.

» The type $\texttt{Circ}(A, B)$ and the circuit combinators are

- opaque, non-programmable
- flow of gates classically fixed

Trying to build circuit combinators

» requires the non-available quantum control

» quantum control known to not play well with classical control

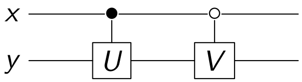
Plan

The first successful attempt at implementing a quantum test:

$$\texttt{input}\ x, y \vdash \texttt{qif}^\circ\ x\ \texttt{then}\ (x, U\,y)\ \texttt{else}\ (x, V\,y)$$

Perform U or V on y conditionally on x without measuring.

» A naïve compilation approach would do



» if U and V are “orthogonal”, one can get rid of x

» The orthogonality property is limited, hard to state

» But QML compiles down to circuits: fully quantum [Altenkirch&Grattage,2005]

van Tonder's Quantum λ-Calculus

Programs in superposition: [vanTonder,2004]

van Tonder defines a syntactic λ-calculus with

- λ-terms stored in quantum registers

$$\Big|(\lambda mn.\lambda xy.mx(nxy))(\lambda xy.x(xy))(\lambda xy.x(x(x(xy))))\Big\rangle$$

- β-reduction as unitary operation
- Constants such as 0, 1 and H:

$$|H\,0\rangle \quad\longrightarrow\quad \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

The unitarity constraints are too strong

- The terms in superpositions are morally the same
- Turning the language into a purely classical one

Linear algebraic lambda-calculi

A side track to overcome the issue

[Arrighi&Dowek,2008],[DiazCaro&al]

- Allow linear combinations of terms (aka "superposition")
  - $\lambda x.M$ is an operator where M can be a linear combination
  - $N(\alpha V + \beta W) \to \alpha(NV) + \beta(NW)$
- Relax the constraints on orthogonality and norm

Advantages

- Full power of λ-calculus
- The β-reduction works fine
- Isolate and study separately problems and solutions

Inconvenient (for this talk)

- Not completely quantum anymore:
  No unitarity nor compilation to circuits

Plan

Approach using Realizability

Based on the linear algebraic lambda-calculus

- Types defined as set of values
- `qbit` defined as

  “normalized superpositions of `true` and `false`”
- $A \to B$ defined as

  “all M such that whenever N realizes A, MN realizes B”
- Types organically emerge from the operational semantics

Discussion

- Capture both quantum control and classical control
- But unitarity is a global property of the term
- and no clear correspondance with physical hardware

Another approach: Reversible Pattern-Matching

Reversible pattern matching [Sabry,V,Vizzotto, 2018]

- syntax for circuits with type constructors $\oplus$ and $\otimes$
- tests using pattern-matching
- $\mathtt{Circ}(a, b)$ becomes programmable: we use $a \leftrightarrow b$ instead

Following circuit-description languages we add

- recursive types, e.g. $[a] \equiv 1 \oplus (a \otimes [a])$
- higher-order on isos :
  - iso-variables
  - boxes in circuits can be iso-variables
  - lambda-abstractions $\lambda f.\{\cdots\}$ and application
  - fixpoints : $\mu f.\{\cdots\}$
  - operational semantics: substitution and unfolding

Termination

- Fixpoints are required to terminate on all inputs

An iso describes a bijective map on the sets of values

Example of "complex" program: the map operation

Let $f : a \leftrightarrow b$.
Define `map` $f : [a] \leftrightarrow [b]$ as

$$\mu \mathbf{g}^{[a]\leftrightarrow[b]}. \left\{ \begin{array}{lll} [] & \leftrightarrow & [] \\ h : t & \leftrightarrow & \texttt{do} \begin{array}{l} h \text{ — } \boxed{f} \text{ — } h' \\ t \text{ — } \boxed{\mathbf{g}} \text{ — } t' \end{array} \; \texttt{return}\ h' : t' \end{array} \right\}$$

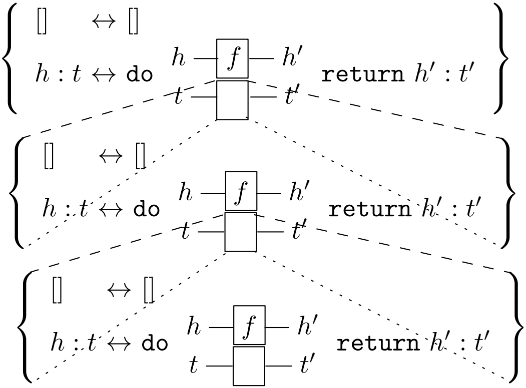
The combinator `map` is typed with

$$(a \leftrightarrow b) \to ([a] \leftrightarrow [b])$$

Another approach: Reversible Pattern-Matching

Example of "complex" program: the map operation

Let $f : a \leftrightarrow b$.
Define `map` $f : [a] \leftrightarrow [b]$ as



Isos as Unitary Maps in $\ell^2(a)$

$\ell^2(\texttt{[Bool]})$

- Hilbert space
- Basis: all possible lists of Boolean values

For example

Consider map Had of type $\texttt{[Bool]} \leftrightarrow \texttt{[Bool]}$ defined as

$$\mu \mathbf{g}^{\texttt{[Bool]}\leftrightarrow\texttt{[Bool]}}.\left\{\begin{array}{lcl} \texttt{[]} & \leftrightarrow & \texttt{[]} \\ h : t & \leftrightarrow & \texttt{do}\ \begin{array}{l} h \;—\boxed{\texttt{Had}}—\; h' \\ t \;—\boxed{\mathbf{g}}—\; t' \end{array}\ \texttt{return}\ h' : t' \end{array}\right\}$$

with

$$\texttt{Had} = \left\{\begin{array}{lcl} \texttt{true} & \leftrightarrow & \frac{1}{\sqrt{2}} \cdot \texttt{true} + \frac{1}{\sqrt{2}} \cdot \texttt{false} \\ \texttt{false} & \leftrightarrow & \frac{1}{\sqrt{2}} \cdot \texttt{true} - \frac{1}{\sqrt{2}} \cdot \texttt{false} \end{array}\right\}$$

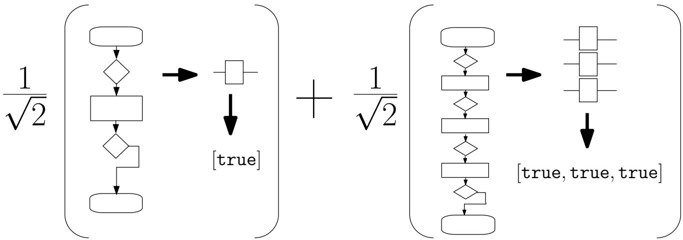
Isos as Unitary Maps in $\ell^2(a)$

For example

Apply $\texttt{map}\,\texttt{Had}$ of type $[\texttt{Bool}] \leftrightarrow [\texttt{Bool}]$ on

$$\frac{1}{\sqrt{2}}[\texttt{true}] + \frac{1}{\sqrt{2}}[\texttt{true}, \texttt{true}, \texttt{true}]$$

and get



A syntactic superposition of executions

State of the Union

- Quantum control is on the way!
- Curry-Howard correspondence in progress
  [Chardonnet,Saurin,V,2023] [Chardonnet,Lemmonier,V,2023]
- Interaction quantum/classical in progress
  [Dave,Lemonnier,Péchoux,Zandzhiev,2025]
- Efficient compilation process in progress
  [Hainry,Péchoux,Silva, 2023+2024]
- Expressive type systems (dependent, etc) still missing.

Plan

Conclusion: Quantum Programming Language

Practical aspect

- Coding quantum algorithms!
- Design choices for quantum circuit-description
- Monadic approach, amenable to oracle synthesis

Theoretical aspect

- Foundational work: quantum lambda-calculus
- Type system based on linear logic
- Extensions to capture circuit-description
- Expressing quantum control still WIP

Questions?