ΝΑΝΟΗΛΕΚΤΡΟΝΙΚΗ & ΚΒΑΝΤΙΚΕΣ ΠΥΛΕΣ

8η Διάλεξη

Βιβλιογραφία:

1. Foundations of Quantum Programming, Mingsheng Ying (1st edition, Elsevier 2016).

2. The Quipper Language,
   http://www.mathstat.dal.ca/~selinger/quipper/

This presentation is based on the contributed article: *Programming the Quantum Future, Benoît Valiron, Neil J. Ross, Peter Selinger, D. Scott Alexander, Jonathan M. Smith, Communications of the ACM, Vol. 58 No. 8, Pages 52-61, DOI: 10.1145/2699415.*

http://cacm.acm.org/magazines/2015/8/189851-programming-the-quantum-future/abstract

The Quipper language is a programming framework for quantum computation

- *Quipper is an embedded, scalable functional programming language for quantum computing.*
- *The Quipper system is a compiler rather than an interpreter; it translates a complete program all in one go rather than executing statements one by one.*
- *The output of the compiler consists of quantum circuits: networks of interconnected, reversible logic gates.*
- *A circuit can take the form of a wiring diagram, but it also constitutes a sequence of instructions ready to be executed by suitable quantum hardware or a simulator.*

Quipper is implemented as an Embedded Domain-Specific Language (EDSL) inside the host language Haskell

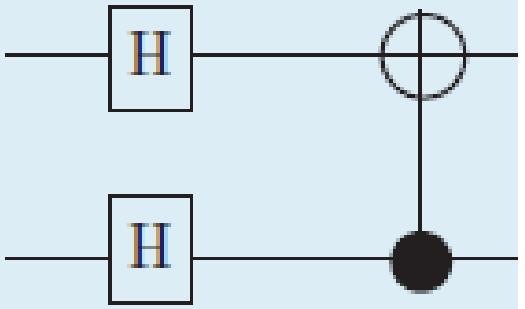
Its main features are:

- Hardware independence. *Quipper's paradigm is to view quantum computation at the level of logical circuits.*
- Extended circuit model. *The initialization and termination of qubits is explicitly tracked for the purpose of ancilla management.*
- Hierarchical circuits. *Quipper features subroutines at the circuit level, permitting a compact representation of circuits in memory.*
- Two runtimes. *The 1st runtime is "circuit generation," and the 2nd runtime is "circuit execution."*
- Parameter/input distinction. *Quipper has two notions of classical data: "parameters," which must be known at circuit-generation time, and "inputs," which may be known only at circuit-execution time.*
- Automatic generation of quantum oracles. *Quantum algorithms require some nontrivial classical computation to be made reversible and then lifted to quantum operation. Quipper has facilities for turning an ordinary Haskell program into a reversible circuit. This is implemented using a "Template Haskell" feature.*

Quipper Feature Highlights with code examples

1) Procedural paradigm. *Qubits are held in variables, and gates are applied one at a time:*

```
mycirc :: Qubit -> Qubit -> Circ (Qubit, Qubit)
mycirc a b = do
  a <- hadamard a
  b <- hadamard b
  (a,b) <- controlled_not a b
  return (a,b)
```

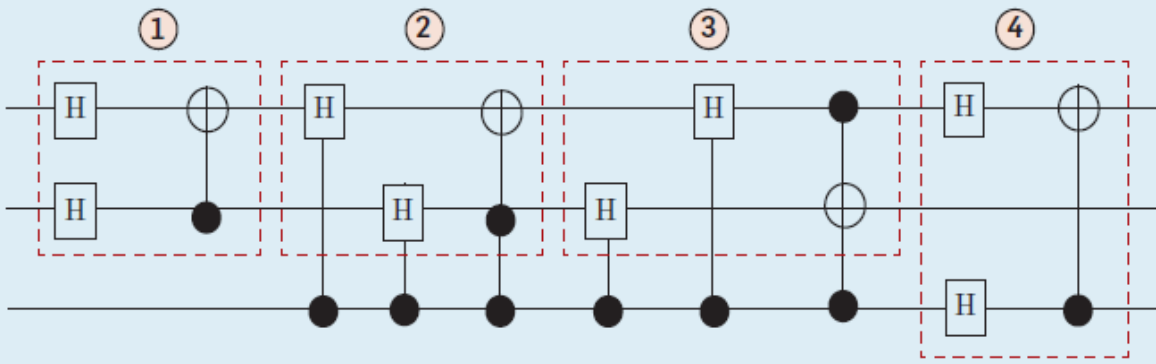


- The type of a circuit-producing function is distinguished by the keyword ***Circ*** after the arrow.
- The function ***mycirc*** inputs *a* and *b* of type Qubit and outputs a pair of qubits while generating a circuit.

Quipper Feature Highlights with code examples

2) Block structure. *Functions generating circuits can be reused as subroutines to generate larger circuits:*

```
mycirc2 :: Qubit -> Qubit -> Qubit -> Circ (Qubit, Qubit, Qubit)
mycirc2 a b c = do
(1) mycirc a b
    with_controls c $ do
(2)   mycirc a b
(3)   mycirc b a
(4) mycirc a c
    return (a,b,c)
```

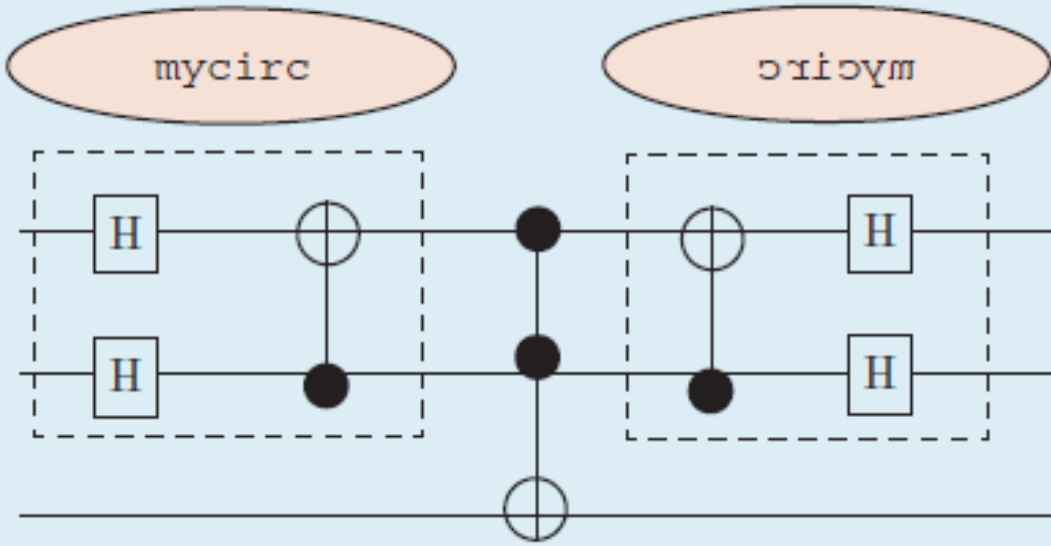


- Operators such as ***with _ controls***, can take an entire block of code as an argument.
- "***do***" introduces an indented block of code.
- "***$***" before "***do***" is an idiosyncrasy of Haskell syntax.

Quipper Feature Highlights with code examples

3) Circuit operators. *Quipper can treat circuits as data and provide high-level operators for manipulating whole circuits:*

```
timestep :: Qubit -> Qubit -> Qubit -> Circ (Qubit, Qubit, Qubit)
timestep a b c = do
    mycirc a b
    qnot c `controlled` (a,b)
    reverse_simple mycirc (a,b)
    return (a,b,c)
```

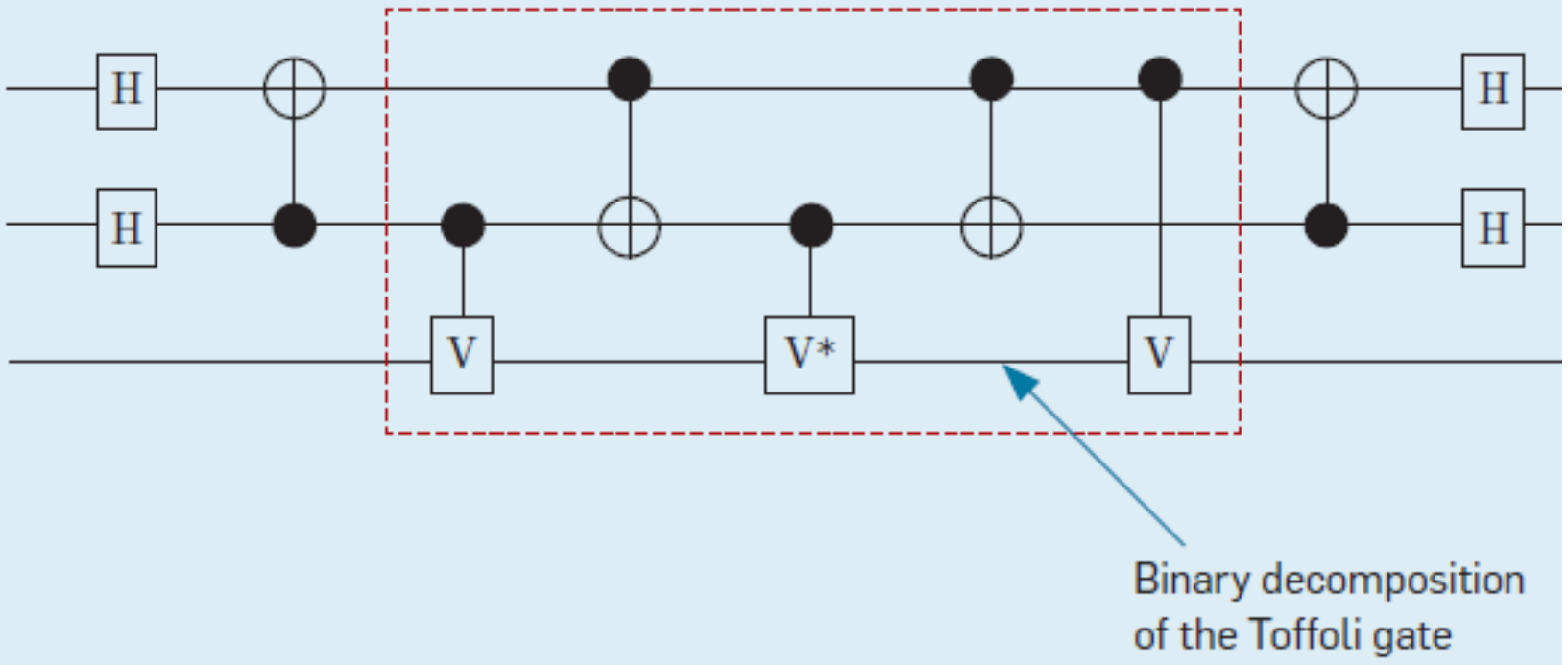


- the operator ***reverse_ simple*** reverses a circuit.

Quipper Feature Highlights with code examples

4) Circuit transformers. *Provides user-programmable "circuit transformers" as a mechanism for modifying a circuit on a gate-by-gate basis:*

```
timestep2 :: Qubit -> Qubit -> Qubit -> Circ (Qubit, Qubit, Qubit)
timestep2 = decompose_generic Binary timestep
```

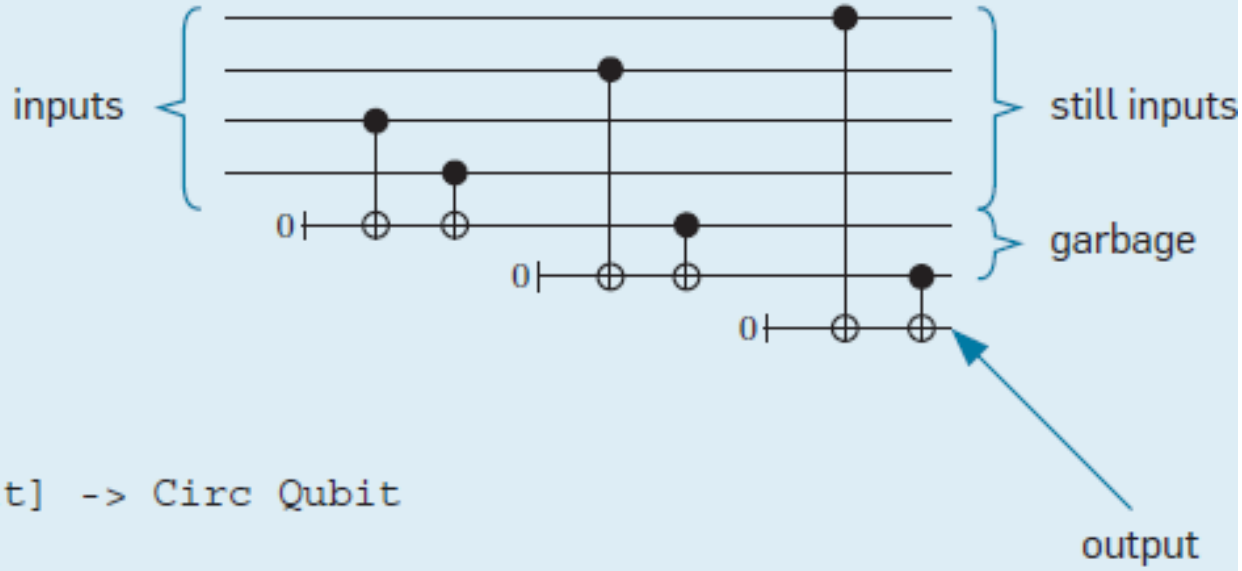


- the ***timestep*** circuit in previous Figure can be decomposed into binary gates using the "***Binary timestep"*** transformer.

Quipper Feature Highlights with code examples

5) Automated functional-to-reversible translation. *Provides a special keyword **build_circuit** for automatically synthesizing a circuit from an ordinary functional program:*

```
build_circuit
f :: [Bool] -> Bool
f as = case as of
  [] -> False
  [h] -> h
  h:t -> h `bool_xor` f t

unpack template_f :: [Qubit] -> Circ Qubit
```

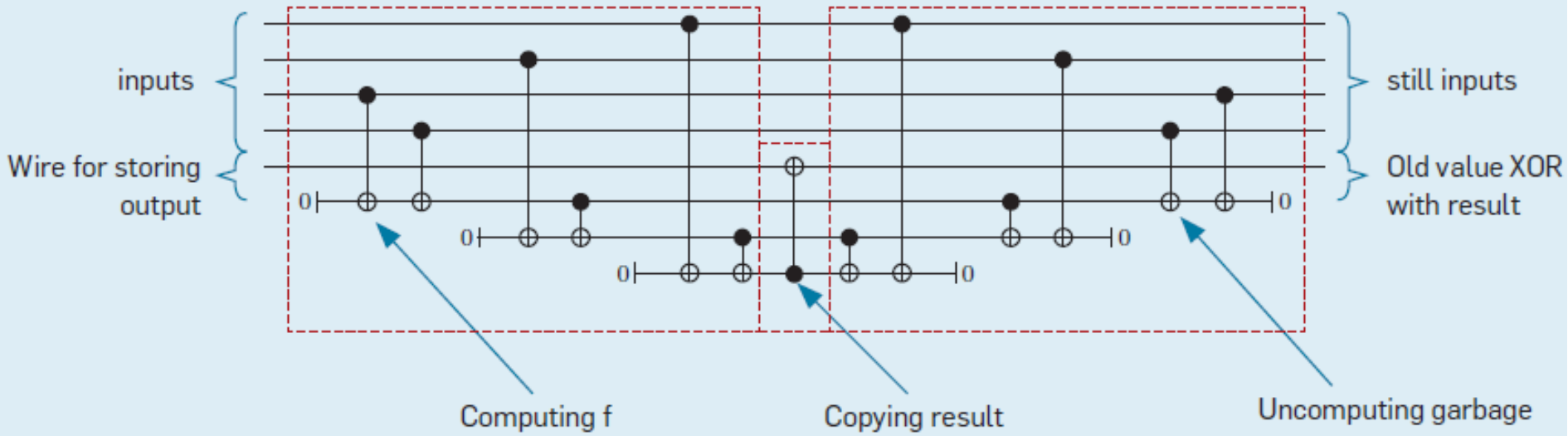


- the type ***Bool*** is used for Boolean "parameters," which must be known at circuit-generation time,
- the type ***Bit*** is used for Boolean "inputs," which may be known only at circuit-execution time.

Quipper Feature Highlights with code examples

5) Automated functional-to-reversible translation. *The resulting circuit from previous Figure can be made reversible with the operator* ***classical _to _ reversible****:*

```
classical_to_reversible :: (Datable a, QCData b) => (a -> Circ b) -> (a,b) -> Circ (a,b)
classical_to_reversible (unpack template_f)
```



Experience with Quipper

Quipper has been used to implement seven nontrivial quantum algorithms, covering a broad spectrum of quantum techniques (*Quantum Fourier Transform, phase estimation, Trotterization, amplitude amplification*):

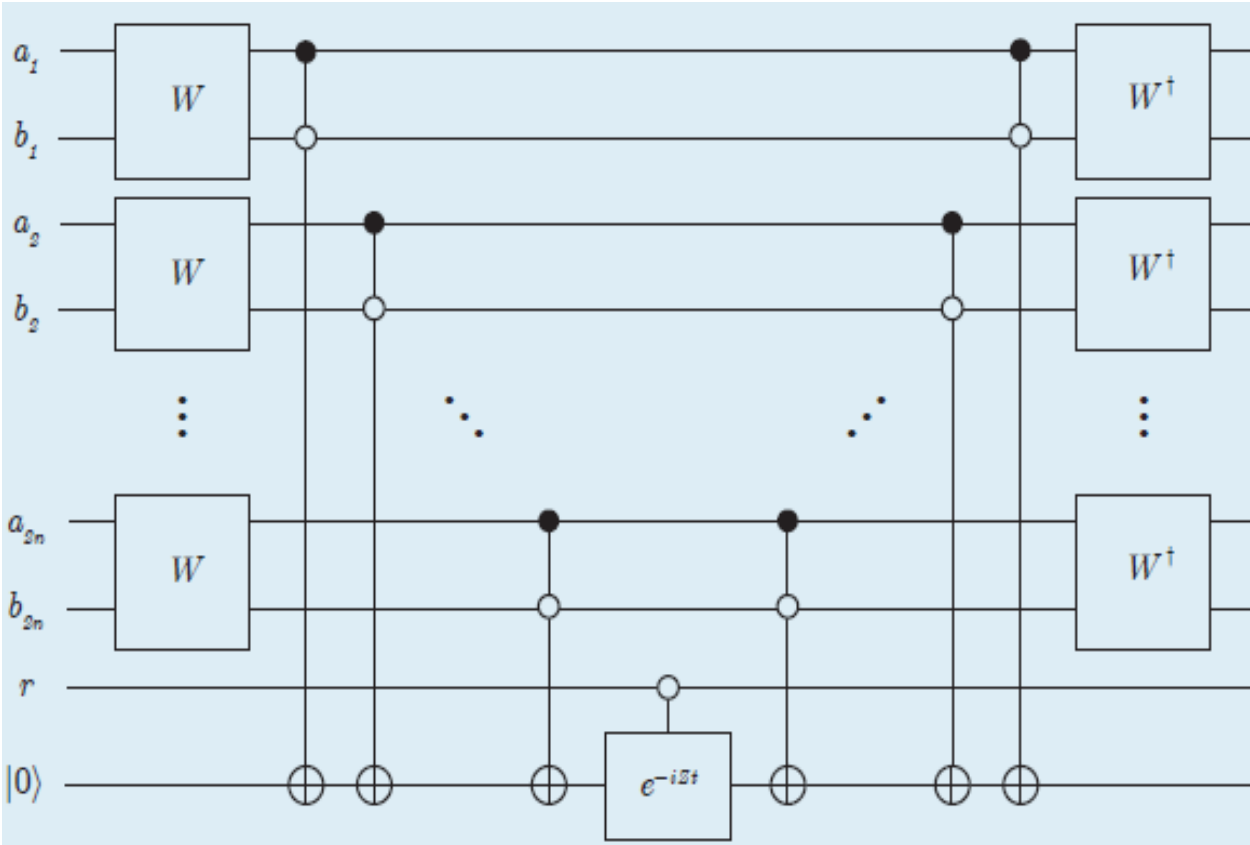
Algorithm	Description
Binary Welded Tree[4]	Finds a labeled node in a graph by performing a quantum walk
Boolean Formula[1]	Evaluates an exponentially large Boolean formula using quantum simulation; QCS version computes a winning strategy for the game of Hex
Class Number[8]	Approximates the class group of a real quadratic number field
Ground State Estimation[19]	Computes the ground state energy level of a molecule
Quantum Linear Systems[9]	Solves a very large but sparse system of linear equations
Unique Shortest Vector[15]	Finds the shortest vector in an *n*-dimensional lattice
Triangle Finding[12]	Finds the unique triangle inside a very large dense graph

- Each of these algorithms solves a problem believed to be classically hard.
- Each algorithm gives an asymptotic quantum speedup, but not necessarily an exponential one.
- All of these algorithms can be run, in the sense that we can print the corresponding *circuits* for *small parameters* and perform *automated gate counts* for circuits *of less tractable sizes*.

Example subroutines written in Quipper

Procedural example. This circuit "implements" the time step for a quantum walk in the *Binary Welded Tree* algorithm (by "implementing" an algorithm, we mean realizing it as a computer program):

1) Inputs a list of pairs of qubits (ai, bi), and a single qubit r.
2) First generates an ancilla qubit in state $|0\rangle$.
3) Applies the two qubit gate W to each of the pairs (a_i, b_i).
4) It is followed by a series of doubly controlled NOT-gates acting on the ancilla.
5) After a middle gate e^{iZt}, it applies all the gates in reverse order.
6) The ancilla ends up in the state $|0\rangle$ and is no longer needed.



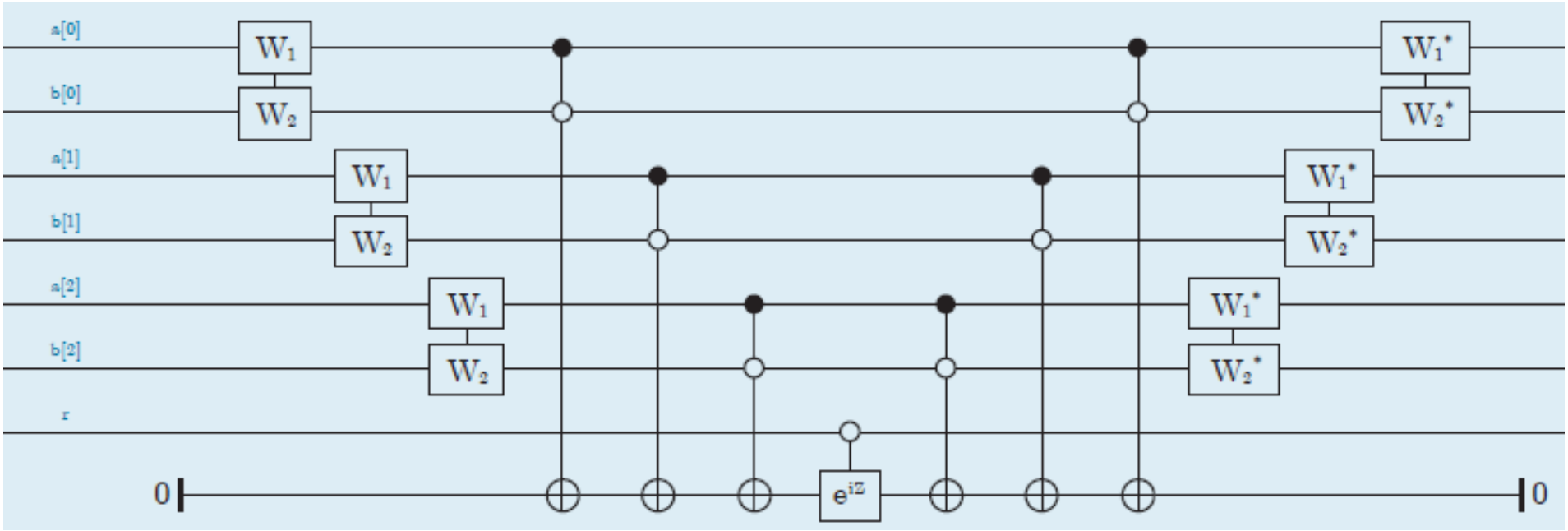
Example subroutines written in Quipper

Procedural example. In the *Binary Welded Tree* algorithm, the Quipper code is:

```
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")
   with_ancilla $ \d -> do
   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
```

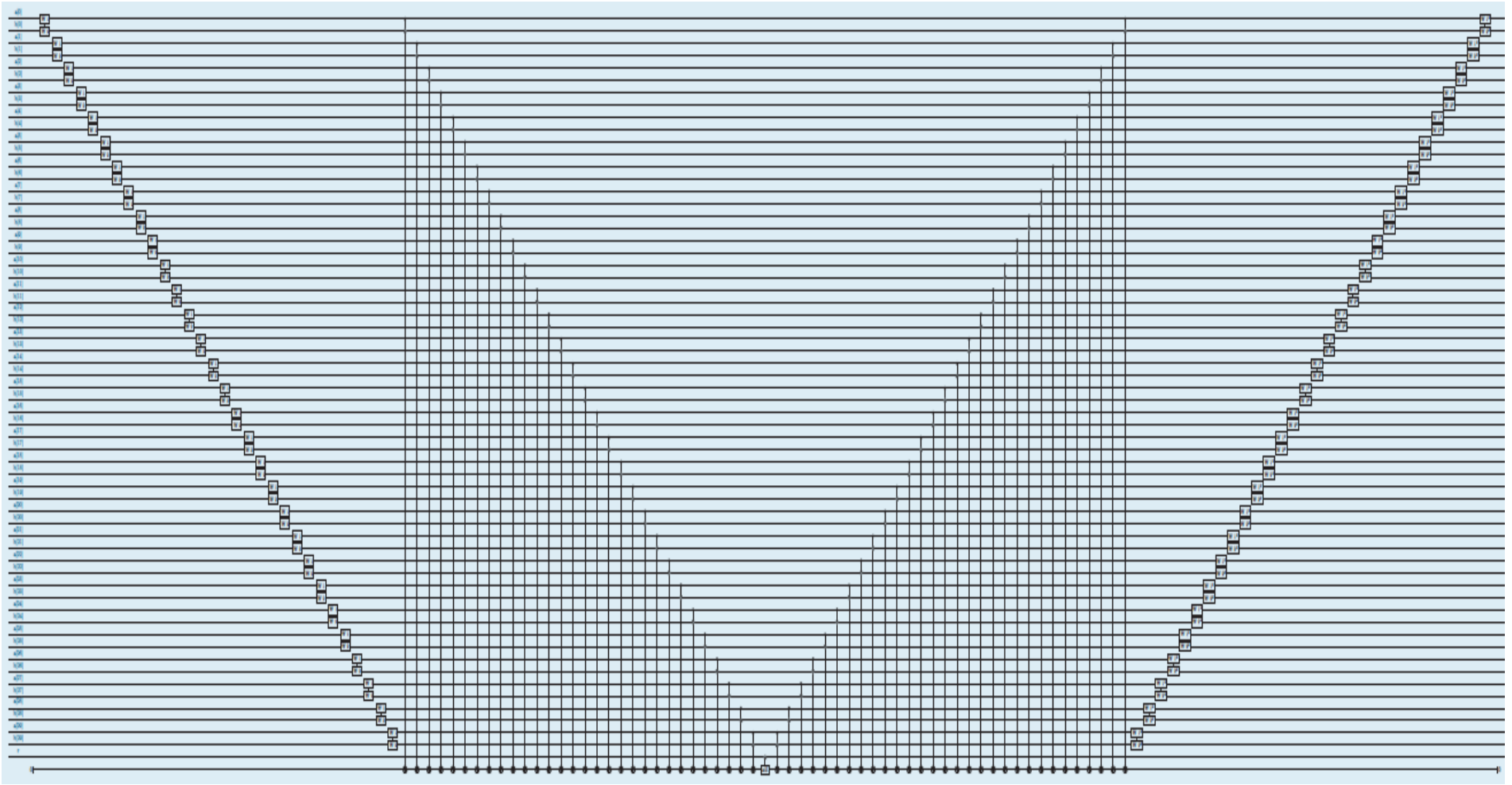
Example subroutines written in Quipper

Procedural example. *The circuit generated by the code in previous Figure, with three qubit pairs:*



Example subroutines written in Quipper

Procedural example. *The circuit generated by the code in previous Figure, with 30 qubit pairs:*



Example subroutines written in Quipper

A functional-to-reversible translation. *Among other things, this algorithm contains an oracle calculating a vector* ***r*** *of complex numbers. The* calcRweights *function is the core of the oracle:*

Contains:

1) Algebraic and transcendental operations on real and complex numbers (such as sin, cos, sinc, and mkPolar).

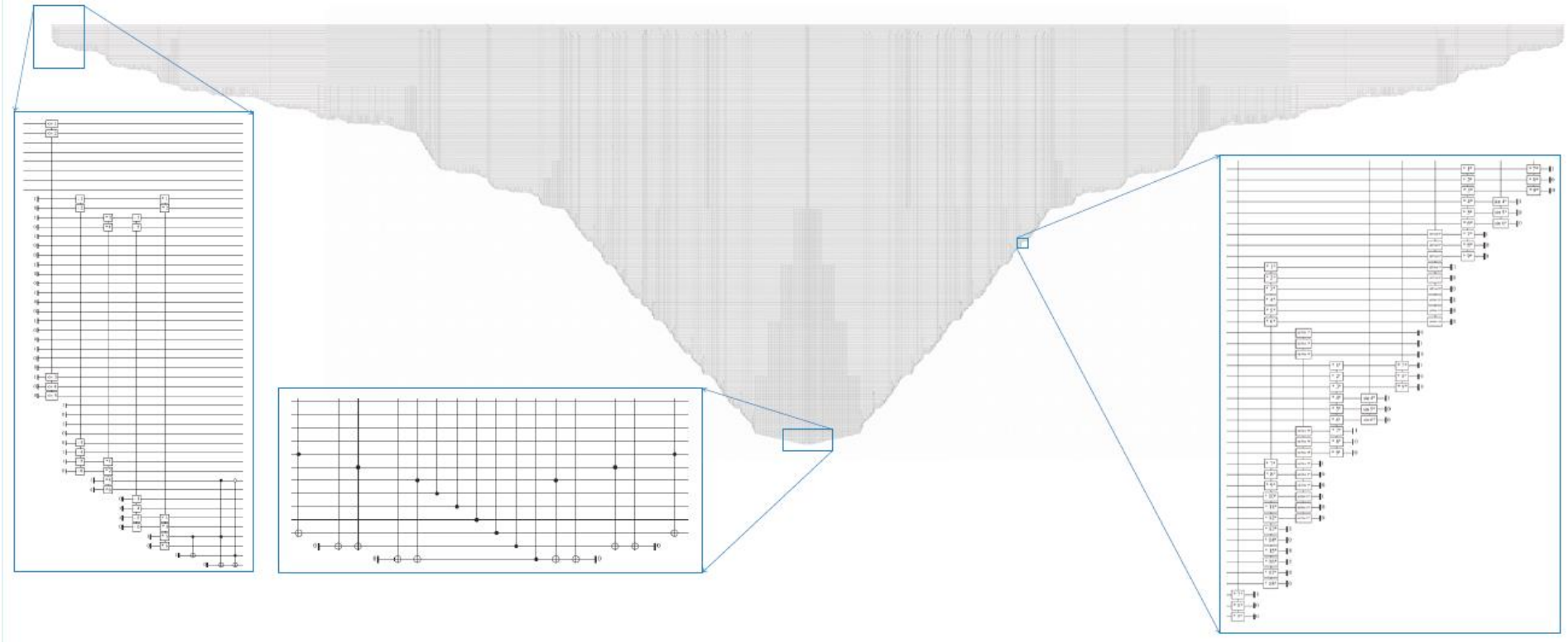
2) Subroutines (such as *edgetoxy* and *itoxy*) not shown in Figure.

3) This Function is readily processed using Quipper's automated circuit-generation facilities.

```
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
```

Example subroutines written in Quipper

A functional-to-reversible translation. *Algebraic and transcendental functions are mapped automatically to quantum versions provided by an existing Quipper library for fixed-point real and complex arithmetic. The result of* calcRweight *function is the rather large circuit:*



This example is from the Quantum Linear Systems algorithm published by *Harrow, A.W., Hassidim, A., and Lloyd, S. Quantum algorithm for solving linear systems of equations. Physical Review Letters 103, 15 (Oct. 2009), 150502.*

The design of Quipper

1. *Quipper is an example of a language suited to a quantum coprocessor model.*
2. *As an embedded language, Quipper is confined to using Haskell's type system, providing many important safety guarantees.*
3. *However, due to Haskell's lack of support for linear types, some safety properties (such as the absence of attempts to clone quantum information) are not adequately supported.*

Note that the absence of cloning is already guaranteed by the physics, regardless of what the programming language does.

However, one could similarly say the absence of an illegal memory access is guaranteed by a classical processor's page-fault mechanism. It is nevertheless desirable to have a programming language that can guarantee *prior to running the program* that the compiled program will never attempt to access an illegal memory location or, in the case of a quantum programming language, *will not attempt to apply a controlled-not gate* to qubits *n* and *m,* where *n = m.*