Reversible circuit compilation with space constraints

Martin Roetteler
Quantum Architectures and Computation Group (QuArC)
Microsoft Research

Based on joint work with Matt Amy, Alex Parent, and Krysta M. Svore:

arXiv:1510.00377    arxiv:1603.01635

QPL 2016
Glasgow, June 9, 2016
Microsoft QuArC and StationQ
Quantum programming in LIQUi|
LIQui|⟩ goals

• Simulation:
  – High enough level language to easily implement large quantum algorithms
  – Allow as large a simulation on classical computers as possible
  – Support abstraction and visualization to help the user
  – Implement as an extensible platform so users can tailor to their own requirements

• Compilation:
  – Multi-level analysis of circuits to allow many types of optimization
  – Circuit re-writing for specific needs (e.g., different gate sets, noise modeling)
  – Compilation into real target architectures
A software architecture for quantum computing

- **Goal:** automatically translate quantum algorithm to executable code for a quantum computer

- **Increases speed of innovation**
  - Rapid development of quantum algorithms
  - Efficient testing of architectural designs
  - Flexible for the future

---

**The LIQUi|> platform**

- Quantum Algorithms
- Programming Language
- Quantum Circuits
- Compilers and Optimizers
- Optimized Quantum Circuits
- Hardware Backend
- Simulation Backend

Wecker and Svore, 2014
The LIQUi|> simulation platform

- **We chose F# as high-level language for quantum algorithms**
  - F# is also the implementation language.
- **Optimized simulation of quantum operations**
  - Parallelized linear algebra package.
  - Many higher-level optimizations are implemented, such as growing a complex circuit into a single multi-qubit unitary operation.
  - A CHP-based stabilizer simulator is included for algorithms that don’t require full circuit and state vector simulation.
- **Public release for academic purposes**
  - Restricted to 23 qubits for circuit simulation.
  - No software restrictions on the stabilizer simulator.

**LIQUi|>**: A Software Design Architecture and Domain-Specific Language for Quantum Computing. Dave Wecker, Krysta M. Svore

Languages, compilers, and computer-aided design tools will be essential for scalable quantum computing, which promises an exponential leap in our ability to execute complex tasks. LIQUi|> is a modular software architecture designed to control quantum hardware. It enables easy programming, compilation, and simulation of quantum algorithms and circuits, and is independent of a specific quantum architecture. LIQUi|> contains an embedded, domain-specific language designed for programming quantum algorithms, with F# as the host language. It also allows the extraction of a circuit data structure that can be used for optimization, rendering, or translation. The circuit can also be exported to external hardware and software environments. Two different simulation environments are available to the user which allow a trade-off between number of qubits and class of operations. LIQUi|> has been implemented on a wide range of runtimes as back-ends with a single user front-end. We describe the significant components of the design architecture and how to express any given quantum algorithm.


**Software:** [http://stationq.github.io/Liquid](http://stationq.github.io/Liquid)
First coding challenge just ended

Interested in delving into quantum chemistry, linear algebra, teleportation, and much more? Students entered the Microsoft Quantum Challenge to see how far they could go! From around the world students investigated and solved problems facing the quantum universe using Microsoft's simulator, LIQUi|>.

They won big prizes, and the opportunity to visit Microsoft Research and maybe gain an internship.

Winners

We are delighted to announce the winners of the Challenge. Interest over the past three months came from all around the world. The judging panel was impressed by all the entries. The following were chosen to receive prizes. Congratulations to the winners!

Each of the winners used the simulator for Language-Integrated Quantum Operations: LIQUi|> from Microsoft Research. Read more on our Blog.

Thien Nguyen
Research School of Engineering, Australian National University, Canberra, Australia
Grand Prize - $5,000
Entry: Simulating Dynamical Input-Output Quantum Systems with LIQUi|>

New links
- Enjoy the blog Announcing the Winners
- Read the winning entries on GitHub

Deadlines
- Launch: February 1, 2016
- Submissions close: April 29, 2016
- Announcement of winners: May 16, 2016

The Challenge is now closed.

Official Rules
Read the Official Rules

Links
- Register for the Challenge
- Read the FAQ for answers
- Learn about LIQUi|>
- Watch short videos about LIQUi|>
- Watch the tutorial video
- Discover the QuArC Group
- Download the simulator
Quantum “Hello World!”

- Define a function to generate entanglement:
  ```
  let EPR (qs:Qubits) = H qs; CNOT qs
  ```

- The rest of the algorithm:
  ```
  let teleport (qs:Qubits) =
    let qs' = qs.Tail
    EPR qs'; CNOT qs; H qs
    M qs'; BC X qs'
    M qs; BC Z !!(qs,0,2)
  ```
Teleport: running the code

loop N times:
  ... create 3 qubits
  ... init the first one to a random state
  ... print it out
  teleport qs
  ... print out the result

```
0:0000.0/Initial State: (  0.3735-0.2531i)|0>+(-0.4615-0.7639i)|1>
0:0000.0/Final State: (  0.3735-0.2531i)|0>+(-0.4615-0.7639i)|1> (bits:10)
0:0000.0/Initial State: ( -0.1105+0.3395i)|0>+(-0.927+0.1146i)|1>
0:0000.0/Final State: ( -0.1105+0.3395i)|0>+(-0.927+0.1146i)|1> (bits:11)
0:0000.0/Initial State: ( -0.3882-0.2646i)|0>+(-0.8092+0.3528i)|1>
0:0000.0/Final State: ( -0.3882-0.2646i)|0>+(-0.8092+0.3528i)|1> (bits:01)
0:0000.0/Initial State: (  0.2336+0.4446i)|0>+(-0.8527+0.1435i)|1>
0:0000.0/Final State: (  0.2336+0.4446i)|0>+(-0.8527+0.1435i)|1> (bits:10)
0:0000.0/Initial State: (  0.9698+0.2302i)|0>+(0.03692+0.0717i)|1>
0:0000.0/Final State: (  0.9698+0.2302i)|0>+(0.03692+0.0717i)|1> (bits:11)
0:0000.0/Initial State: ( -0.334-0.3354i)|0>+(0.315-0.8226i)|1>
0:0000.0/Final State: ( -0.334-0.3354i)|0>+(0.315-0.8226i)|1> (bits:01)
```
More complex circuits

```
let entangle (qs:Qubits) =
  H qs; let q0 = qs.Head
  for q in qs.Tail do CNOT[q0;q]
  M >>= qs
```

```
0:0000.0/#### Iter 0 [  0.2030]: 0000000000000
0:0000.0/#### Iter 1 [  0.1186]: 0000000000000
0:0000.0/#### Iter 2 [  0.0895]: 0000000000000
0:0000.0/#### Iter 3 [  0.0749]: 0000000000000
0:0000.0/#### Iter 4 [  0.0664]: 1111111111111
0:0000.0/#### Iter 5 [  0.0597]: 0000000000000
0:0000.0/#### Iter 6 [  0.0550]: 1111111111111
0:0000.0/#### Iter 7 [  0.0512]: 0000000000000
0:0000.0/#### Iter 8 [  0.0484]: 0000000000000
0:0000.0/#### Iter 9 [  0.0463]: 0000000000000
0:0000.0/#### Iter 10 [  0.0446]: 0000000000000
0:0000.0/#### Iter 11 [  0.0432]: 1111111111111
0:0000.0/#### Iter 12 [  0.0420]: 0000000000000
0:0000.0/#### Iter 13 [  0.0410]: 0000000000000
0:0000.0/#### Iter 14 [  0.0402]: 0000000000000
0:0000.0/#### Iter 15 [  0.0399]: 0000000000000
0:0000.0/#### Iter 16 [  0.0392]: 1111111111111
0:0000.0/#### Iter 17 [  0.0387]: 1111111111111
0:0000.0/#### Iter 18 [  0.0380]: 0000000000000
0:0000.0/#### Iter 19 [  0.0374]: 1111111111111
```
User defined gates

/// <summary>
/// Controlled NOT gate
/// </summary>
/// <param name="qs"> Use first two qubits for gate</param>
[<LQD>]
let CNOT (qs:Qubits) =
    let gate =
        Gate.Build("CNOT",fun () ->
            new Gate(
                Name  = "CNOT",
                Help  = "Controlled NOT",
                Mat   = CSMat(4,[(0,0,1.,0.);(1,1,1.,0.);
                                 (2,3,1.,0.);(3,2,1.,0.)]),
                Draw  = "\ctrl\{#1\}\go\{#1\}\targ"
            )
        )
    gate.Run qs
Full teleport circuit in a Steane7 code
Shor’s algorithm component: modular adder

As defined in:  
Circuit for Shor’s algorithm using 2n+3 qubits  
– Stéphane Beauregard

let op (qs:Qubits) =  
CCAdd a cbs  // Add a to Φ|b⟩  
AddA' N bs  // Sub N from Φ|a + b⟩  
QFT' bs  // Inverse QFT of Φ|a + b − N⟩  
CNOT [bMx;anc]  // Save top bit in Ancilla  
QFT bs  // QFT of a+b-N  
CAAdd A N (anc :: bs)  // Add back N if negative  
CCAdd' a cbs  // Subtract a from Φ|a + b mod N⟩  
QFT' bs  // Inverse QFT  
CNOT [bMx]  // Flip top bit  
CAAdd A N (anc :: bs)  // Reset Ancilla to |0⟩  
X [bMx]  // Flip top bit back  
QFT bs  // QFT back  
CCAdd a cbs  // Finally get Φ|a + b mod N⟩
Shor’s algorithm: full circuit: 4 bits \(\cong 8200\) gates

Largest Dave has done:
14 bits (factoring 8189)
14 Million Gates
30 days
Shor’s algorithm: scaling

![Graph showing scaling of Shor’s algorithm](image-url)
LIQUi|→ - Optimizations

• If we can guarantee that the qubits we want to operate on are always at the beginning of the state vector, we can view the operation as:

\[ G_{2k,2k} \otimes I_{2n-k,2n-k} \times \Psi_{2n} \]

• However, what we’d really like is to flip the Kronecker product order:

\[ I_{2n-k,2n-k} \otimes G_{2k,2k} \times \Psi_{2n} \]

• This would accomplish:
  
  – \( I \otimes G \) would become a block diagonal matrix that just has copies of \( G \) down the diagonal. This means that you’d never have to actually materialize \( U = I \otimes G \)
  
  – Processing would be highly parallel (and/or distributed) because the matrix is perfectly partitioned and applies to separate, independent parts of the state vector
Quantum Chemistry

Can quantum chemistry be performed on a small quantum computer? D. Wecker, M. B. Hastings, M. Troyer

As quantum computers will almost certainly provide some practical advantage to simulate molecular systems, it is important to determine how large molecules can be performed. The diagram shows several papers addressing this question:

- **First paper**: ~850 thousand years to solve ($N^9$ scaling)
- **Second paper**: ~30 years to solve ($N^7$ scaling)
- **Third paper**: ~5 days to solve ($N^{5.5}$ scaling)
- **Fourth paper**: ~1 hour to solve ($N^3, Z^{2.5}$ scaling)

Ferredoxin ($Fe_2S_2$) used in many metabolic reactions including energy transport in photosynthesis.

- Intractable on a classical computer
- Assumed quantum scaling: ~24 billion years ($N^{11}$ scaling)
- First paper: ~850 thousand years to solve ($N^9$ scaling)
- Second paper: ~30 years to solve ($N^7$ scaling)
- Third paper: ~5 days to solve ($N^{5.5}$ scaling)
- Fourth paper: ~1 hour to solve ($N^3, Z^{2.5}$ scaling)

The simulation of molecules is a widely anticipated application of quantum computers. However, as quantum computers become available, it is important to understand the limits of what can be simulated. This question has been addressed in several papers, showing that quantum computers can simulate molecules that are much larger than what can be simulated classically. The scaling of quantum algorithms is crucial to understanding the limits of quantum simulations. However, we find that the scaling of known algorithms can be very poor for large molecules. For example, in the simulation of a molecule with 1000 atoms, the number of gates required is approximately $N^{11}$. This suggests that the scaling of known algorithms is not sufficient for simulating large molecules. Therefore, we developed new algorithms that can simulate larger molecules. Specifically, we developed algorithms that can simulate molecules with 1000 atoms in a small number of gates. This suggests that quantum computers can simulate molecules that are much larger than what can be simulated classically.
Quantum Chemistry

\[ H = \sum_{pq} h_{pq} a_p^\dagger a_q + \frac{1}{2} \sum_{pqrs} h_{pqrs} a_p^\dagger a_q^\dagger a_r a_s \]
Quantum and reversible circuit synthesis
Instruction sets: universal single-qubit bases

- $T + \text{Clifford} \ (H, X, Y, Z, I, S)$
  
  $$T = R \left( \frac{\pi}{4} \right) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$$

- $V_3 + \text{Clifford} \ (H, X, Y, Z, I, S)$
  
  $$V_3 = \frac{1}{\sqrt{5}} \begin{bmatrix} 1 + 2i & 0 \\ 0 & 1 - 2i \end{bmatrix}$$

- $\frac{\pi}{12} + \text{Clifford} \ (H, X, Y, Z, I, S)$
  
  $$R \left( \frac{\pi}{6} \right) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/6} \end{bmatrix}$$

- Fibonacci anyon basis:
  
  $$\sigma_1 = \begin{bmatrix} -\omega & 0 \\ 0 & \omega^3 \end{bmatrix}, \sigma_2 = \begin{bmatrix} \omega^4\tau & -\omega^2\sqrt{\tau} \\ -\omega^2\sqrt{\tau} & -\tau \end{bmatrix},$$
  
  $$\omega = e^{i\pi/5}, \tau = \frac{\sqrt{5} - 1}{2}$$
Quantum compiling

Quantum algorithm

Quantum computer

Reversible

Error correction

\[ \approx HTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHTHT
Year 2012: Revolution in synthesis methods!
based on algebraic number theory

Single qubit unitary over $\mathbb{C}$

Single qubit unitary over $\mathbb{Z}[i, \frac{1}{\sqrt{2}}]$

Single qubit Clifford + T circuit

Unitary round off procedure

Single qubit exact synthesis algorithm

Number of T gates required is $O\left(\log \left(\frac{1}{\varepsilon}\right)\right)$ vs $O\left(\log^{3+\delta} \left(\frac{1}{\varepsilon}\right)\right)$
(for the Solovay-Kitaev algorithm)

[Kliuchnikov/Maslov/Mosca’12], [Selinger’12], [Ross/Selinger’14], [Kliuchnikov/Yard’15]
Reversible computing: why bother?

- **Arithmetic:**
  - Factoring: just needs “constant” modular arithmetic
  - ECC dlogs: need generic modular arithmetic
  - HHL: need integer inverses; Newton type methods

- **Amplitude amplification:**
  - Implementation of the “oracles”, e.g., for search, collision etc.
  - Implementation of walk operators on data structures

- **Quantum simulation:**
  - Addressing/indexing functions for sparse matrices
  - Computing Hamiltonian terms on the fly

See also: “lifting monad” in Quipper
Universal gate set: Toffoli gates

**Fact:** The set \{Toffoli, CNOT, NOT\} is universal for reversible computing: any *even* permutation on n qubits can be written as a sequence of Toffoli, CNOT, and NOT gates. [Toffoli’80], [Fredkin/Toffoli’82]

**Main motivation:** How can we find efficient implementations of reversible circuits in terms of efficient Toffoli networks?
How can we do this starting from irreversible descriptions in a programming language like Python or Haskell or F# or C?
Can we trade time (circuit depth) for space (#qubits) in a meaningful way?
Example: Carry ripple adder (in F#)

```fsharp
let carryRippleAdder (a:bool []) (b:bool []) =
    let n = Array.length a
    let result = Array.zeroCreate (n)
    result.[0] <- a.[0] <> b.[0]
    let mutable carry = a.[0] && b.[0]
    result.[1] <- a.[1] <> b.[1] <> carry
    for i in 2 .. n - 1 do
        // compute outgoing carry from current bits and incoming carry
        carry <- (a.[i-1] && (carry <> b.[i-1])) <> (carry && b.[i-1])
        result.[i] <- a.[i] <> b.[i] <> carry
    result
```
Example: If-then-else expressions

// module emission_tst_workaround: float -> float -> unit
// author = MG_Burns, changeset = 1519992, date = 06/03/2009

let THRTTL_MIN = 1.0
let THRTTL_MAX = 49.9

let emission_tst_workaround (v_front_wheels:float) (v_rear_wheels:float) =
  let epa_detect = (v_front_wheels > 0.0) && (v_rear_wheels = 0.0)
  if epa_detect then
    let throttleSettings = THRTTL_MIN
    let catConverterOn = true
  else
    let throttleSettings = THRTTL_MAX
    let catConverterOn = false
  runEngine throttleSettings catConverterOn

// MGB: just like taking candy from a baby
If-then-else construct I

\[ |x\rangle \rightarrow \text{pred} \rightarrow |y\rangle \quad \text{if} \quad \text{pred}^{-1} |0\rangle \rightarrow |y'\rangle \quad \text{else} \]
If-then-else construct II

\[
\begin{aligned}
&|x\rangle & &|x\rangle \\
&|0\rangle & &|0\rangle \\
&|0\rangle & &|0\rangle \\
&\text{pred} & &\text{pred}^{-1} \\
&|y\rangle & &|y\rangle \\
&|1\rangle & &|1\rangle \\
&\text{B} & &\text{A} \\
\end{aligned}
\]

[Maslov, Saeedi '01]
If-then-else construct III

\[
\begin{align*}
|x\rangle & \quad \text{pred} \quad |x\rangle \\
|0\rangle & \quad |0\rangle \\
|0\rangle & \quad |0\rangle \\
\end{align*}
\]

\[
\begin{align*}
|y\rangle & \quad A \quad |y\rangle \\
|0\rangle & \quad B \quad |0\rangle \\
|0\rangle & \quad B^{-1} \quad |y'\rangle \\
\end{align*}
\]

\[
\begin{align*}
|0\rangle & \quad A^{-1} \quad |0\rangle \\
\end{align*}
\]
Reversible computing: at the gate level

• We assume that function is given as **combinational circuits**, i.e., circuits that do not make use of memory elements or feedback.

• Universal families of irreversible gates:

  ![Diagram](image)

  • We can compose gates together to make larger circuits.

• Basic issue: many gates are **not** reversible!
Reversible computing: at the gate level

Example:

Replace each gate with a reversible one: (e.g. = Toffoli gate)
Cleaning up the scratch bits

Replace each gate with a reversible one [Bennett, IBM JRD’73]:
Pebble game: case of 1D graph

Rules of the game: [Bennett, SIAM J. Comp., 1989]

- n boxes, labeled i = 1, ..., n
- in each move, either add or remove a pebble
- a pebble can be added or removed in i=1 at any time
- a pebble can be added or removed in i>1 if and only if there is a pebble in i-1
- 1D nature arises from decomposing a computation into “stages”

Example:

```
1     2     3     4
1     1     2     2
3     3     4     4
5     3     6     2
7     1
```
Pebble game: 1D plus space constraints

Imposing resource constraints:
• only a total of $S$ pebbles are allowed
• corresponds to reversible algorithm with at most $S$ ancilla qubits

Example: ($n=3$, $S=3$)

<table>
<thead>
<tr>
<th>#</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
</tbody>
</table>
Optimal pebbling strategies

**Definition:** Let $X$ be solution of pebble game. Let $T(X)$ be the number of steps and $S(X)$ be the number of pebbles. Define $F(n, S) = \min \{ T(X) : S(X) \leq S \}$.

### Table (small values of $F$):

<table>
<thead>
<tr>
<th>$n$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>9</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>11</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>15</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>17</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>9</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>19</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>21</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>11</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>23</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>12</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>25</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>14</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>29</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>15</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>31</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>29</td>
<td>29</td>
</tr>
<tr>
<td>16</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>33</td>
<td>31</td>
<td>31</td>
<td>31</td>
<td>31</td>
</tr>
<tr>
<td>17</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>35</td>
<td>33</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>18</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>37</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>19</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td></td>
</tr>
</tbody>
</table>

6/9/2016

M. Roetteler @ MSR / QuArC

[E.Knill, arxiv:math/9508218]
**Dynamic programming:** Allowed us to find best strategy for given number of steps $n$ to be performed and given space resource constraint $S$ which is the number of available pebbles.

This works ok for 1D chains. For general graphs the problem of finding the optimal strategy is difficult (PSPACE complete problem) -> need heuristics
Optimal pebbling strategies: 1D chains

[Bennett '73]

[Lange-McKenzie-Tapp 2000]
Time-space tradeoffs

Let A be an algorithm with time complexity $T$ and space complexity $S$.

- Using reversible pebble game, [Bennett, SIAM J. Comp. 1989] showed that for any $\varepsilon > 0$ there is a reversible algorithm with time $O(T^{1+\varepsilon})$ and space complexity $O(S \ln(T))$.

- Issue: one cannot simply take the limit $\varepsilon \to 0$. The space would grow in an unbounded way (as $O(\varepsilon^{2/\varepsilon} S \ln(T))$).

- Improved analysis [Levine, Sherman, SIAM J. Comp. 1990] showed that for any $\varepsilon > 0$ there is a reversible algorithm time $O(T^{1+\varepsilon}/S^{\varepsilon})$ and space complexity $O(S (1+\ln(T/S)))$.

- Other time/space tradeoffs: [Buhrman, Tromp, Vitányi, ICALP’01]
  \[ T_{rev} = S 3^k 2^{O(T/2^k)}, \quad S_{rev} = O(kS), \]  where $k =$ #pebbles
  special cases: $k = O(1) \to$ [Lange-McKenzie-Tapp, 2000]
  $k = \log T \to$ [Bennett, 1989]

- Pebble games played on general DAGs hard to analyze (opt #pebbles = PSPACE complete) → need heuristics to tackle general dependency graphs!
New technique: Mutable data flow analysis
Mutability via in-place operations: e.g. adders

• This is an example for in-place operation \((x, y) \rightarrow (x, x+y)\)
• At the program level, mutable data can be identified (e.g. via `mutable`)

Manufacturing more in-place computations

Out-of-place circuit for $f$:

\[ U_f \]

$|x\rangle \quad |0\rangle \quad |x\rangle \quad |f(x)\rangle$

**Generic circuit identity:** [Kashefi et al], [Mosca et al] describe method that allows in-place efficient computation of $f$, provided that the inverse has an efficient circuit too.

\[ U_f \quad |x\rangle \quad |0\rangle \quad U_{f^{-1}} \quad |f(x)\rangle \quad |0\rangle \]
Mutable data dependency graph (MDD)

Example:

```latex
let f a b = a \&\& b
```

Corresponding circuit:

Corresponding MDD:
Mutable data dependency graph (MDD)

Example: function inlining; Boolean ops

```
let f a b =
a || b
let g a b =
a && b
let h a b c d =
f a b <> g c d
```

Corresponding MDD (only graph for f is shown; similar for g, h)
Example (cont’d)

Note: - all ancilla qubits (scratch bits) are returned back in the 0 state (indicated by “|”)
  - Some ancilla qubits are reused in the circuit (red circles above)
  - Leads to space savings and offers advantage over alternative methods (e.g. original Bennett)
Algorithm to clean up qubits early

**Algorithm 2** EAGER Performs eager clean-up of an MDD.

**Require:** An MDD $G$ in reverse topological order, subroutines LastDependentNode, ModificationPath

1: $i \leftarrow 0$
2: **for each** node in $G$ **do**
3: \hspace{1em} **if** modificationArrows node $= \emptyset$ **then**
4: \hspace{2em} dIndex $\leftarrow$ LastDependentNode of node in $G$
5: \hspace{2em} path $\leftarrow$ ModificationPath of node in $G$
6: \hspace{2em} input $\leftarrow$ InputNodes of path in $G$
7: \hspace{2em} **if** None (modificationArrows input) $\geq$ dIndex **then**
8: \hspace{3em} cleanUp $\leftarrow$ (Reverse path) ++ cleanNode
9: \hspace{2em} **end if**
10: \hspace{1em} **else**
11: \hspace{2em} cleanUp $\leftarrow$ uncleanNode
12: \hspace{2em} $G \leftarrow$ Insert cleanUp Into $G$ After dIndex
13: \hspace{2em} **end if**
14: **end for**
15: **return** $G$
REVS: Examples
An example at scale: SHA-2

Hash function:

Initialize hash values
h0 := 0x6a09e667
h1 := 0xbb67ae85
...
h7 := 0x5be0cd19

Initialize constants
k[0..63] := 0x428a2f98, 0x71374491, 0xb5c0fbcf, ...

Do preprocessing
break message into 512-bit chunks (16 32bit ints)

Expand to 64 32 bit ints as follows:
Create W: a 64 entry array of 32 bit ints
Copy the message into w[0..15] and do:
for each chunk
  for i from 16 to 63
    s0 := (w[i-15] ≪ 7) ⊕ (w[i-15] ≪ 18) ⊕ (w[i-15] ≪ 3)
    s1 := (w[i-2] ≪ 17) ⊕ (w[i-2] ≪ 19) ⊕ (w[i-2] rshift 10)
    w[i] := w[i-16] + s0 + w[i-7] + s1

Initialize working variables to current hash value:
a := h0
...
h := h7

Compression function main loop:
Do compression rounds
Add the compressed chunk to the current hash value:
h0 := h0 + a
...
h7 := h7 + h

digest := hash := h0 :: h1 :: h2 :: h3 :: h4 :: h5 :: h6 :: h7
Example: SHA-2 (in F#)

```
let hash x =
    let a = x.[0..31], b = x.[32..63], c = x.[64..95],
        d = x.[96..127], e = x.[128..159], f = x.[160..191],
        g = x.[192..223], h = x.[224..255]
    (%modAdd 32) (ch e f g) h
    (%modAdd 32) (s0 a) h
    (%modAdd 32) w h
    (%modAdd 32) k h
    (%modAdd 32) h d
    (%modAdd 32) (ma a b c) h
    (%modAdd 32) (s1 e) h
    for i in 0 .. n - 1 do
        hash (rot 32*i x)
```
The SHA-2 round function

- The function operates on a 512-bit message, divided into 16 32-bit chunks.
- Each chunk is processed through a series of operations involving constants, message chunks, and Boolean functions.
- The constants and message chunks are combined with the previous round's output through addition modulo 2^32.
- The Boolean functions, including Ch, Σ0, Σ1, and Ma, transform the intermediate values in a non-linear manner.

Each 32-bit wide message chunk is processed through a network of operations, resulting in the output of the SHA-2 round function.
SHA-2: hand-optimized reversible circuit
SHA-2: comparing different cleanup methods

<table>
<thead>
<tr>
<th>Rounds</th>
<th>#qubits</th>
<th>Toffoli count</th>
<th>time</th>
<th>#qubits</th>
<th>Toffoli count</th>
<th>time</th>
<th>#qubits</th>
<th>Toffoli count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>704</td>
<td>1124</td>
<td>0.2546002</td>
<td>353</td>
<td>690</td>
<td>0.3290822</td>
<td>353</td>
<td>683</td>
</tr>
<tr>
<td>2</td>
<td>832</td>
<td>2248</td>
<td>0.2639522</td>
<td>353</td>
<td>1380</td>
<td>0.3360352</td>
<td>353</td>
<td>1366</td>
</tr>
<tr>
<td>3</td>
<td>960</td>
<td>3372</td>
<td>0.2823012</td>
<td>353</td>
<td>2070</td>
<td>0.3420732</td>
<td>353</td>
<td>2049</td>
</tr>
<tr>
<td>4</td>
<td>1088</td>
<td>4496</td>
<td>0.2827132</td>
<td>353</td>
<td>2760</td>
<td>0.3543582</td>
<td>353</td>
<td>2732</td>
</tr>
<tr>
<td>5</td>
<td>1216</td>
<td>5620</td>
<td>0.2907102</td>
<td>353</td>
<td>3450</td>
<td>0.3664272</td>
<td>353</td>
<td>3415</td>
</tr>
<tr>
<td>6</td>
<td>1344</td>
<td>6744</td>
<td>0.3042492</td>
<td>353</td>
<td>4140</td>
<td>0.3784522</td>
<td>353</td>
<td>4098</td>
</tr>
<tr>
<td>7</td>
<td>1472</td>
<td>7868</td>
<td>0.3123962</td>
<td>353</td>
<td>4830</td>
<td>0.3918812</td>
<td>353</td>
<td>4781</td>
</tr>
<tr>
<td>8</td>
<td>1600</td>
<td>8992</td>
<td>0.3284542</td>
<td>353</td>
<td>5520</td>
<td>0.4025412</td>
<td>353</td>
<td>5464</td>
</tr>
<tr>
<td>9</td>
<td>1728</td>
<td>10116</td>
<td>0.3341342</td>
<td>353</td>
<td>6210</td>
<td>0.4130702</td>
<td>353</td>
<td>6147</td>
</tr>
<tr>
<td>10</td>
<td>1856</td>
<td>11240</td>
<td>0.3449002</td>
<td>353</td>
<td>6900</td>
<td>0.4304762</td>
<td>353</td>
<td>6830</td>
</tr>
</tbody>
</table>

All timings measured running the F# compiler in VS 2013 on an Intel i7-3667 @ 2Ghz 8GB RAM (6 cores) under Win 8.1
We’re beating many REVLIB benchmarks

<table>
<thead>
<tr>
<th>name</th>
<th>Our Method</th>
<th>RevLib</th>
<th>Comparison (rel.)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>4mod5</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>5xp1</td>
<td>23</td>
<td>6</td>
<td>83</td>
<td>23</td>
</tr>
<tr>
<td>6sym</td>
<td>11</td>
<td>4</td>
<td>35</td>
<td>14</td>
</tr>
<tr>
<td>alu4</td>
<td>61</td>
<td>39</td>
<td>2821</td>
<td>33</td>
</tr>
<tr>
<td>apex5</td>
<td>228</td>
<td>23</td>
<td>3727</td>
<td>1025</td>
</tr>
<tr>
<td>bw</td>
<td>36</td>
<td>3</td>
<td>73</td>
<td>87</td>
</tr>
<tr>
<td>con1</td>
<td>13</td>
<td>4</td>
<td>16</td>
<td>13</td>
</tr>
<tr>
<td>decod24</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>e64</td>
<td>193</td>
<td>63</td>
<td>4096</td>
<td>195</td>
</tr>
<tr>
<td>ex1010</td>
<td>38</td>
<td>18</td>
<td>6581</td>
<td>29</td>
</tr>
<tr>
<td>f51m</td>
<td>52</td>
<td>30</td>
<td>1774</td>
<td>35</td>
</tr>
<tr>
<td>frg2</td>
<td>336</td>
<td>54</td>
<td>8950</td>
<td>1219</td>
</tr>
<tr>
<td>hwb9</td>
<td>33</td>
<td>15</td>
<td>2915</td>
<td>170</td>
</tr>
<tr>
<td>max46</td>
<td>20</td>
<td>10</td>
<td>195</td>
<td>17</td>
</tr>
<tr>
<td>mini-alu</td>
<td>9</td>
<td>3</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>pdc</td>
<td>102</td>
<td>46</td>
<td>3222</td>
<td>619</td>
</tr>
<tr>
<td>rd84</td>
<td>26</td>
<td>14</td>
<td>170</td>
<td>34</td>
</tr>
<tr>
<td>seq</td>
<td>107</td>
<td>31</td>
<td>3310</td>
<td>1617</td>
</tr>
<tr>
<td>spla</td>
<td>95</td>
<td>33</td>
<td>3232</td>
<td>489</td>
</tr>
<tr>
<td>sqrt8</td>
<td>18</td>
<td>6</td>
<td>32</td>
<td>18</td>
</tr>
<tr>
<td>squar5</td>
<td>16</td>
<td>3</td>
<td>36</td>
<td>17</td>
</tr>
<tr>
<td>t481</td>
<td>19</td>
<td>2</td>
<td>26</td>
<td>20</td>
</tr>
</tbody>
</table>

**Bold** = we beat in size + width

**Normal** = we beat in width
Simulating Toffoli networks is easy

```ocaml
type Primitive =
  | RTOFF of int * int * int
  | RCNOT of int * int
  | RNOT of int

let simCircuit (gates:Primitive list) (numberOfBits:int) (input:bool list) =
  let bits = Array.init numberOfBits (fun _ -> false)
  List.iteri (fun i elm -> bits.[i] <- elm) input
  let applyGate gate =
    match gate with
    | RNOT a -> bits.[a] <- not bits.[a]
    | RCNOT(a, b) -> bits.[b] <- bits.[b] <> bits.[a]
    | RTOFF(a, b, c) -> bits.[c] <- bits.[c] <> (bits.[a] && bits.[b])
  List.iter applyGate gates
  bits
```

2/19/2016  M. Roetteler @ MSR / QuArC  57
Compiler verification
Why verify?

How do we know that these are indeed the outputs of the circuit?
Simulating Toffoli networks is easy

Reversible Toffoli network computing (?) a SHA-2 hash function
with 353 bits, 3334 gates
Generated by Revs & rendered by LIQui||
ReVer

An irreversible program to reversible circuit compiler, implemented and verified in F* (https://www.fstar-lang.org/)

What that **does** mean:

- The program interpreter and compiled circuit produce the same output
- Compiled circuits return all ancillas to their initial state

What that **doesn’t** mean:

- That the compiled program is correct
- That the F* proof checker is correct
- The the compiled circuit will produce the same output for every interpreter/hardware
ReVer: Operational semantics

\[
\begin{align*}
\textbf{Store} \quad \sigma : \mathbb{N} &\rightarrow \mathbb{H} \\
\textbf{Config} \quad c := (t, \sigma) &\quad \textbf{REFL} \quad \langle v, \sigma \rangle \Rightarrow \langle v, \sigma \rangle \\
\quad \langle t_1, \sigma \rangle \Rightarrow \langle \lambda x. t'_1, \sigma' \rangle &\quad \langle t_2, \sigma' \rangle \Rightarrow \langle v_2, \sigma'' \rangle \\
\quad \langle t'_1[x \mapsto v_2], \sigma'' \rangle \Rightarrow \langle v, \sigma''' \rangle &\quad \langle (l_1, t_2), \sigma \rangle \Rightarrow \langle v, \sigma''' \rangle \\
\textbf{APP} &\quad \langle (l_1, t_2), \sigma \rangle \Rightarrow \langle v, \sigma''' \rangle \\
\quad \langle t_1, \sigma \rangle \Rightarrow \langle l_1.t_1, \sigma' \rangle &\quad \langle t_2, \sigma' \rangle \Rightarrow \langle l_2, \sigma'' \rangle \\
\quad \langle t_1 \leftarrow t_2, \sigma \rangle \Rightarrow \langle \text{unit}, \sigma''[t_1 \mapsto \sigma''(l_2)] \rangle &\quad \langle t_1 \times t_2, \sigma \rangle \Rightarrow \langle l_3, \sigma''[l_3 \mapsto \sigma''(l_1 \times \sigma''(l_2))] \rangle \\
\textbf{ASSN} &\quad \langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma' \rangle \\
\quad \langle t_2, \sigma' \rangle \Rightarrow \langle l_2, \sigma'' \rangle &\quad i \not\in \text{dom}(\sigma) \\
\quad \langle t_1 \leftarrow t_2, \sigma \rangle \Rightarrow \langle \text{register } l_1 \ldots l_m, \sigma' \rangle &\quad \langle t_2, \sigma' \rangle \Rightarrow \langle \text{register } l_{m+1} \ldots l_n, \sigma'' \rangle \\
\textbf{BEXP} &\quad \langle t_1, \sigma \rangle \Rightarrow \langle \text{true}, \sigma'' \rangle \\
\quad \langle t_2, \sigma' \rangle \Rightarrow \langle \text{false}, \sigma'' \rangle &\quad \langle \text{false}, \sigma \rangle \Rightarrow \langle l, \sigma''[l \mapsto 0] \rangle \\
\quad \langle t_1 \leftarrow t_2, \sigma \rangle \Rightarrow \langle \text{register } l_1 \ldots l_m, \sigma' \rangle &\quad \langle \text{register } l_{m+1} \ldots l_n, \sigma'' \rangle \\
\textbf{INDEX} &\quad \langle i, \sigma \rangle \Rightarrow \langle l_i, \sigma' \rangle \\
\quad 1 \leq i \leq n &\quad \langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma_1 \rangle \\
\quad \langle t_2, \sigma \rangle \Rightarrow \langle l_2, \sigma_2 \rangle &\quad \vdots \\
\quad \langle t, \sigma \rangle \Rightarrow \langle \text{register } l_1 \ldots l_m, \sigma' \rangle &\quad \langle t_n, \sigma \rangle \Rightarrow \langle l_n, \sigma_n \rangle \\
\quad 1 \leq i \leq j \leq n &\quad \langle \text{register } l_1 \ldots l_n, \sigma \rangle \Rightarrow \langle \text{register } l_1 \ldots l_n, \sigma_n \rangle \\
\textbf{SLICE} &\quad \langle t, \sigma \rangle \Rightarrow \langle \text{register } l_1 \ldots l_m, \sigma' \rangle \\
\quad 1 \leq i \leq j \leq n &\quad \langle \text{rotate } t i, \sigma \rangle \Rightarrow \langle \text{register } l_1 \ldots l_{i-1}, \sigma' \rangle \\
\quad \langle t[i..j], \sigma \rangle \Rightarrow \langle \text{register } l_i \ldots l_j, \sigma'' \rangle &\quad \langle \text{rotate } t i, \sigma \rangle \Rightarrow \langle \text{register } l_i \ldots l_{i-1}, \sigma' \rangle \\
\textbf{ROTATE} &\quad \langle t, \sigma \rangle \Rightarrow \langle \text{register } l_1 \ldots l_m, \sigma' \rangle \\
\quad 1 < i < n &\quad \langle \text{clean } t i, \sigma \rangle \Rightarrow \langle \text{unit}, \sigma''[\text{dom}(\sigma' \setminus \{i\})] \rangle \\
\quad \langle t, \sigma \rangle \Rightarrow \langle l, \sigma' \rangle &\quad \langle \text{assert } t, \sigma \rangle \Rightarrow \langle \text{unit}, \sigma' \rangle \\
\quad \sigma'(i) = \text{false} &\quad \langle \text{assert } t, \sigma \rangle \Rightarrow \langle \text{unit}, \sigma' \rangle \\
\end{align*}
\]
Circuit compiler and interpreter. Written and verified in F*

Two verified paths:
• Bennett-style compilation, translate directly to circuit
• Space-efficient Boolean expression compilation
THANK YOU!

http://research.microsoft.com/groups/quarc/

LIQ\(U_i\rangle\) is publicly available from
http://stationq.github.io/Liquid

martinro@microsoft.com