Lecture
Quantum Computing
(CS5070)
Introduction to Silq

Professor Dr. rer. nat. habil. Sven Groppe
https://www.ifis.uni-luebeck.de/index.php?id=groppe
Motivation: Circuits versus Control Flow

- **Qiskit**: Developers have to think about how to solve their problems by realizing circuits
  
  ![Qiskit Example](image)

  def ghz():
  ghz_circuit = QuantumCircuit(3)
  ghz_circuit.h(0)
  ghz_circuit.cx(0, 1)
  ghz_circuit.cx(1, 2)
  return ghz_circuit;

  circuit = ghz()

- **Silq**: Express algorithm by control flow of programs
  
  ![Silq Example](image)

  def ghz(){
    a:=0:B;
    b:=0:B;
    c:=0:B;
  a:=H(a);
  if a { b := X(b); }
  if b { c := X(c); }
  return (a,b,c); }

  def main() {
    circuit = ghz();
  }

  more familiar to computer scientists and software developers

  variables and control flow to be mapped to quantum circuits
  
  future work/to be done for running Silq program on real quantum computers

  so far only simulation
Silq: Control Flow

\[ |0\rangle \xrightarrow{\text{x}} |0\otimes n\rangle \xrightarrow{n} x \xrightarrow{H} y \]

\[
\begin{cases}
  x := 0 : \mathbb{B} ; \\
x := 0 : \text{int}[n] ; \\
y := H(x) ; \\
x := H(x) ; \\
\text{if } x \{ \\
  y := H(y) ; \\
\} \\
\text{for } k \text{ in } [0..n) \{ \\
  x[k] := H(x[k]) ; \\
\} 
\end{cases}
\]
Silq: Multi-qubit quantum logic gates 1/2

```python
def CX(const x: B, y: B): B{
    if x {
        y := X(y);
    }
    return y;
}

def CZ(const x: B, y: B): B{
    if x {
        y := Z(y);
    }
    return y;
}

def SWAP(x: B, y: B): B^2{
    return (y, x);
}
```

![Diagram of quantum gates](image)
Silq: Multi-qubit quantum logic gates 2/2

```python
def CCX(const x:B,
        const y:B,z:B):B{
    if x && y {
        z := X(z);
    }
    return z;
}
```

```python
def CSWAP(const x:B,
          y:B,z:B):B^2{
    if x {
        a:=z;
        z:=y;
        y:=a;
    }
    return (y,z);
}
```

```python
def X_CX_X(const x:B,y:B):B{
    if !x {
        y := X(y);
    }
    return y;
}
```
Motivation: Newly Designed Language

- **Qiskit**: Libraries for python, i.e., integrated into existing prog. language
  - Easy start for python developers
  - Usage of existing libraries for e.g. visualization
  - "Inherited" support of jupyter notebooks
  - No static error detection for common quantum computing errors
  - Error-prone manual uncomputation
  - Function calls to construct circuit patterns, but not integrated into language constructs

- **Silq**: Prog. Language especially designed for quantum computing
  - Developers have to learn new programming language
  - Existing libraries cannot be used/no possibility to call existing libraries
  - No support of jupyter notebooks, but integration into Visual Studio Code available
  - Static type checking prevents common quantum computing errors
  - **Automatic** uncomputation via static type checking
  - Easy to learn language constructs instead of complex circuits for features like quantum indexing and support of QRAM
## Supported Types in Silq

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Alt. Symbol</th>
<th>Must be classical</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>1</td>
<td></td>
<td>The singleton type that only contains element ()</td>
</tr>
<tr>
<td>(!\tau)</td>
<td>result</td>
<td></td>
<td>type (\tau), but restricted to classical values</td>
</tr>
<tr>
<td>(\mathbb{B})</td>
<td>B</td>
<td></td>
<td>Booleans, i.e. bits (!(\mathbb{B})) or qubits ((\mathbb{B}))</td>
</tr>
<tr>
<td>(N)</td>
<td>N</td>
<td>✓</td>
<td>Natural numbers 0, 1, ...</td>
</tr>
<tr>
<td>(Z)</td>
<td>Z</td>
<td>✓</td>
<td>Integers ..., -1, 0, 1, ...</td>
</tr>
<tr>
<td>(Q)</td>
<td>Q</td>
<td>✓</td>
<td>Rational numbers</td>
</tr>
<tr>
<td>(\mathbb{R})</td>
<td>R</td>
<td>✓</td>
<td>Reals. Simulation semantics are implementation-defined (typically floating point)</td>
</tr>
<tr>
<td>int[n]</td>
<td></td>
<td></td>
<td>n-bit integers encoded in two's complement</td>
</tr>
<tr>
<td>uint[n]</td>
<td></td>
<td></td>
<td>n-bit unsigned integers</td>
</tr>
<tr>
<td>(\tau\times\cdots\times\tau)</td>
<td>(\tau \times \cdots \times \tau)</td>
<td></td>
<td>tuple types, e.g., (\mathbb{B}\times\text{int}[n])</td>
</tr>
<tr>
<td>(\tau[])</td>
<td></td>
<td></td>
<td>dynamic-length arrays</td>
</tr>
<tr>
<td>(\tau^n)</td>
<td></td>
<td></td>
<td>vectors of length (n)</td>
</tr>
<tr>
<td>[\text{const}] (\tau\times\cdots\times[\text{const}]\ \tau\rightarrow[m\text{free}</td>
<td>q\text{free}]\ \tau)</td>
<td></td>
<td>functions, optionally annotated as mfree or qfree, whose input types are optionally annotated as const</td>
</tr>
</tbody>
</table>

\(n\) stands for an arbitrary expression of type !\(N\)
Annotations - Classical types (!)

- Duplicate classical annotations can be ignored: !!\tau \equiv !\tau
- Classical annotations commute with
  - tuples: !(\tau \times \cdots \times \tau) \equiv !\tau \times \cdots \times !\tau
  \Rightarrow !(\tau \times \tau) \equiv !(\tau \times !\tau) \equiv !(!\tau \times \tau) \equiv !(!\tau \times !\tau)
  - arrays: !\tau[] \equiv (!\tau)[] \equiv !(\tau[])
  - fixed-length arrays: !\tau^n \equiv (!\tau)^n \equiv !(\tau^n)
- Classical values can be re-interpreted as quantum values: !\tau \subseteq \tau (with \tau quantum type)
- No special issues to be considered for functions with only classical values (classical parameters, classical return value and classical variables):

```python
def usingClassicalTypes(x:!B,f:!B!->!B){
    return f(x);  // f is classical
}
```
Annotations - const

- Annotation `const` indicates that a variable will not be changed in the given context
  - Each parameter of a function and each variable in the context may be annotated as `const`
  - We can use constant parameters and variables more liberally, since they are guaranteed to persist in the given context
  - It is assumed that a `const` parameter is consumed from the callee of the function (after the function is called) and not from the function itself
    - consuming = applying gate (changing the quantum state)
    - consuming does not include conditions (not changing the quantum state)

Example:

```python
def myEval(const x: B, f: const B!->B):
    return f(x);
```
Annotations - mfree

- Annotation *mfree* indicates that a function evaluation is guaranteed to be without applying any measurements

Example:

```python
def myEval(f: mfree) mfree(
    return f(false); // => myEval is mfree
}
```
Deferred Measurement Principle

- Measuring commutes with conditioning:
  ⇒ The choice of whether to measure a qubit before, after, or during an operation conditioned on that qubit will have no observable effect on a circuit's final outcomes

- measuring qubits as early as possible can reduce the maximum number of simultaneously stored qubits
  - usage of smaller quantum computers or more efficient simulations

  VERSUS deferring all measurements until the end of circuits:
  → temporary results and ancilla bits can be analyzed

- Measurement of one qubit ⇒ Measurement all of its entangled qubits collapsing to the corresponding basis states ⇒ Side-effects!

- Before never using an ancilla qubit or reusing it, "un-entagle" this qubit for no side-effects ⇒ Uncomputation
Example of **Uncomputation**

- Silq hides use of ancilla qubits and uncomputation:
  
  \[
  d := a \lor b \lor c;
  \]

- Ancilla qubits must be explicitly uncomputed in other languages:

  **Qiskit:**
  
  \[
  c = \text{QuantumCircuit}(5) \\
  c.cx(0,3);c.cx(1,3);c.cx(0,1,3) \\
  c.cx(2,4);c.cx(3,4);c.cx(2,3,4)
  \]

  **Quipper:**
  
  ```
  with_computed (OR a b) $ \\
  \text{\textbackslash t} -> \text{OR \ t c}
  ```

  **Q#:**
  
  ```
  using(t=Qubit()){
  OR(a,b,t); \\
  OR(t,c,d); \\
  Adjoint OR(a,b,t);
  }
  ```
Automatic Uncomputation simplifies Code & reduces Code Size

Silq:

cTri := 0:int[rrbar];
for j in [0..rrbar) {
    for k in [j+1..rrbar) {
        if ee[tau[j]][tau[k]]
            && eew[j] && eew[k] {
                cTri += 1;
            }
    }
}

Quipper:

cTri <- foldM (\cTri j -> do
    let tau_j = tau ! j
    eed <- qinit (intMap_replicate rr False)
        -- computing eed = ee[tau[j]]
    (taub, ee, eed) <- all_FetchE tau_j ee eed
    cTri <- foldM (\cTri k -> do
        let tau_k = tau ! k
        eed_k <- qinit False
        -- eedd_k=eed[tau[k]]=ee[tau[j]][tau[k]]
    (tauc, eed, eedd_k) <- qram_fetch qram tau_k eed eedd_k
        -- using eedd_k as ctrl
    cTri <- increment cTri `controlled` eedd_k .&&. (eew ! j) .&&. (eew ! k)
        -- uncomputing eedd_k
    (tauc, eed, eedd_k) <- qram_fetch qram tau_k eed eedd_k
    qterm False eedd_k
    return cTri)
    cTri [j+1..rrbar-1]
    -- uncomputing eed
    (tauc, ee, eed) <- all_FetchE tau_j ee eed
    qterm (intMap_replicate rr False) eed
    return cTri)
cTri [0..rrbar-1]

QWire:

index : [r: Nat, r: Nat], CMC[\n, \n, 1] → ...
qindex : [r: Nat, r: Nat], CMC[\n, \n, 1] → ...
controlEOM : [r: Nat], CMC[1, 0, 1] → ...

EvalCondition : [r: Nat, r: Nat, r: Nat], CMC[1, 0, 1] → ...
qinit (\rbar [rbar] [rbar]) : rbar : CMC[1, 1, 1] → ...

boxed (ee, tau, eew) =<
    (tau, tau) <- unbox (index rbar) \tau \tau -- tau/tau[i]
    (tau, tau) <- unbox (index rbar) \tau \tau -- tau/tau[i]
    (ew, tau) <== unbox (index rbar rbar) \tau \tau -- unbox
        
        (ew, wav, ewd, k) <== unbox (index rbar rbar) \tau \tau -- unbox
        
        (ew, wav, ewd, k) <== unbox (index rbar rbar) \tau \tau -- unbox
    (ewd, k, ewd, k, wav, c) <== unbox and ewd, k, wav, c
        -- condition
    output (ew, wav, ewd, k, wav, ewd, k, wav, c) -- output

boxed (ee, wav, ewd, c) =<
    (ew, wav, ewd, c) <== unbox (EvalCondition r rbar) \tau \tau -- evaluate
        condition
    (cTri, c) <== unbox (controlledIncrement rbar) cTri c --
        controlled increment
    (ew, wav, ewd) <== unbox (newelemetric EvalCondition r
        rbar) \tau \tau -- unbox
    output (ew, wav, c) -- output
Automatic Uncomputation & Concise Language Constructs reduce Code Size

Q#

-44% code

-50% library

Silq

- Comparison on Microsoft's Q# coding contests, see [https://silq.ethz.ch/comparison](https://silq.ethz.ch/comparison)
Annotations - qfree

- **Annotation qfree**
  - for indicating that evaluating functions or expressions **neither introduces nor destroys superpositions**, 
  - ensures that evaluating qfree functions on classical arguments yields classical results, and
  - enables automatic uncomputation
    - of all temporary values computed in the function.

<table>
<thead>
<tr>
<th>Function</th>
<th>Signature</th>
<th>qfree</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>B → mfree B</td>
<td></td>
<td>H introduces superpositions</td>
</tr>
<tr>
<td>X</td>
<td>B → qfree B</td>
<td>✓</td>
<td>X neither introduces nor destroys superpositions</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>y</td>
<td>const B × const B → qfree B</td>
</tr>
</tbody>
</table>

```python
def myEval(f: B→qfree B) qfree{
    return f(false);
}
```

- myEval takes a qfree function f and evaluates it on false ⇒ myEval itself is also qfree.
Annotations - lifted

- Annotation `lifted` is a shorthand to indicate `qfree` functions with only constant arguments
  - Classical arguments are implicitly treated as constants

Example:

```scala
def MyOr(x: Boolean, y: !Boolean)lifted{ // x and y are implicitly const
    return x || y; // => MyOr is lifted
}
```
Errors detected by Silq's Static Type Checking

- Implicit Measurement
- Conditioned Measurement
- Reverse Measurement
- Using Consumed Variable
- Impossible Uncomputation
Implicit Measurement

• If parameters of functions are not constant, then these parameters must be consumed
  - Not consumed not constant parameter (variables)
    ⇒ trying to forget variable, trying to uncompute variable
      • If automatic uncomputation is not possible and manual forgetting is not specified ⇒ implicit measurement with side-effects ⇒ Silq rejects code

```
def implicitMeas[n:!N](x:uint[n]){
  y := x % 2;
  return y;
} // parameter 'x' is not consumed and cannot be uncomputed
```

• Declaring parameter x to be const ⇒ x has to be consumed or uncomputed by callee of the function

```
def unconsumedConst[n:!N](const x:uint[n]){
  y := x % 2;
  return y;
} // no error
```
Conditioned Measurement

- trying to apply a measurement conditioned on a quantum variable ⇒ **type error**: the then-branch requires a physical action and we cannot determine whether or not we need to carry out the physical action without measuring the condition.

```python
def condMeas(const c:B, x:B){
    if c { x := measure(x); }
    return x;
} // cannot call function 'measure[B]' in 'mfree' context
```

- conditional measurement is possible if `c` is classical:

```python
def classCondMeas(const c:!B, x:B){
    if c { x := measure(x):B; } // `:B` interprets the measurement result as a quantum value
    return x;
} // no error
```

- **error** if measurement is hidden in a passed function, as this function is not `mfree`, i.e., free of measurements:

```python
def hiddenCondMeas(f:B!→B, const c:B, x:B){
    if c { x := f(x); } // error: cannot call function 'f' in 'mfree' context
    return x;
}
Reverse Measurement

- `reverse(f)` returns the inverse of function `f`
- Inverting a measurement would violate quantum mechanics
  ⇒ reverse only operates on mfree functions

```python
def revMeas():
    return reverse(measure);
} // Error: reversed function must be mfree
```

- Inverting `f` is unsafe if `f` is not surjective
  - For example, calling the function returned by `reverse(dup)` is only safe when both its arguments are equal:

```python
def useReverseSafe():
    x:=H(0:BB);
    y:=dup(x); // 1/\sqrt{2} (|00)+|11|)
    reverse(dup[BB])(x,y); // uncomputes y
    return x; // 1/\sqrt{2} (|0|+|1|)
}
```

```python
def useReverseUnsafe():
    x:=H(0:BB);
    y:=H(0:BB); // 1/2 (|00>+|01>+|10>+|11|)
    // UNDEFINED behavior, since dup cannot
    // produce the above state:
    reverse(dup[BB])(x,y);
    return x;
}
```

- `forget(x=y)` is (unsafe) shorthand for `reverse(dup[BB])(x,y)`
Using Consumed Variable

- **Error**: Accessing already consumed variables (which are no longer available after being consumed)

```python
def useConsumed(x: int):
    y := H(x);
    return (x, y); # Error: undefined identifier x
```

- It is **no error** to access already consumed **constant** variables (as they can be duplicated/entangled)

Silent **duplication** of constant $x$:

$$
\begin{array}{cc}
\psi & = & H(x) \\
\end{array}
$$

- All duplicates of constant variables are either consumed (as above), or can be uncomputed
Impossible Uncomputation - Not Constant

```python
def nonConst(y: ℂ):
    if X(y) {  # X consumes y
        phase(π);
    }
    # Error: non-'lifted' quantum expression must be consumed
```

- **While** function `nonConst` **consumes** `y` in `X(y)`, automatic uncomputation (implemented by reversing `X`) **would re-introduce** `y`
- **While** `nonConst` in principle **would work**, it is **disallowed** to prevent this confusing re-introduction of `y`

```python
def signFlipOf0(const y: ℂ):
    if X(y) {  # X consumes a copy of y
        phase(π);
    }
    # no error
```

- **Marking** `y` as `const` clarifies that `y` **should remain in the context** (i.e., the resulting program should be accepted)
- **If** `y` **is const**, `X` **consumes a duplicate of** `y`, thus leaving the original `y` **unchanged**
Impossible Uncomputation - Not qfree

```python
def nonQfree(const y:ℤ, z:ℤ){
    if H(y) {
        z := X(z);
    }
    return z;
} // non-'lifted' quantum expression must be consumed
```

- While function `nonQfree` uses a constant input `y`, automatic uncomputation does not work in this case
  - Intuitively because `H` may introduce additional states into the superposition that cannot be uncomputed in the end (this can be seen by a straight-forward computation)
- To prevent this case, Silq only supports uncomputing `qfree` expressions.
- Of course, uncomputation can always be made explicit by `reverse` or `forget`, at the cost of losing safety
Support of Quantum Indexing

- \( e_1[e_2] \) for non-classical \( e_2 \), if \( e_1 \) does not contain any classical components (i.e., neither classical types, nor function types, nor array types) [Sil Doc.]

- Example for generating the generalized W state for \( 2^k \) qubits:

\[
W_{2^k} = \frac{|100...0\rangle+|010...0\rangle+...+|00...01\rangle}{\sqrt{2^k}}.
\]

```python
def generalizedWState(k:!N){
    i:=0:uint[k];  // for specifying which qubit will be inverted...
    // produce uniform superposition of i over k-bit uints
    for j in [0..k]{ i[j]:=H(i[j]); }
    // for holding the generalized W state:
    qs:=vector(2^k,0: вс);
    // invert i-th qubits (results in correct state, but entangled with i)
    qs[i]=X(qs[i]);  // quantum indexing! Complex quantum circuit necessary for it!
    // if you do not want to return i, then you have to use forget for manual uncomputation of i!
    return (i,qs);
}

def main(){
    return generalizedWState(2);
}
```

Silq-Simulator returns:

\[
\frac{|0,(1,0,0,0))\rangle+|3,(0,0,0,1))\rangle+|2,(0,0,1,0))\rangle+|1,(0,1,0,0))\rangle}{2}
\]
**Manual Uncomputation - forget**

```python
def generalizedWState(k:N){
    i:=0:uint[k]; // for specifying which qubit will be inverted...
    // produce uniform superposition of i over k-bit uints
    for j in [0..k){ i[j]:=H[i[j]]; }
    // for holding the generalized W state:
    qs:=vector(2^k,0:B);
    // invert i-th qubits (results in correct state, but entangled with i)
    qs[i]=X(qs[i]); // quantum indexing! Complex quantum circuit necessary for it!
    // manually uncompute i as it is too complex for automatic uncomputation
    forget(
        // function to reconstruct i from qs,
        // such that uncomputation circuit can be computed for i
        i = λ(qs:B^(2^k)) lifted {
            i:=0:uint[k];
            for j in [0..2^k) {
                if qs[j] { // in the superposition's summand where qs[j]==1, i==j
                    i=j as uint[k];
                }
            }
            return i;
        })(qs)
    );
    return qs;
}

def main(){
    return generalizedWState(2);
}
```
Support of Quantum Memory (QRAM)

• quantum memory is the quantum-mechanical version of classical computer memory
  - classical memory stores information as binary states
  - quantum memory stores a quantum state for later retrieval
  - premature technologies work in laboratory for storing quantum states a "longer" time

• Silq supports QRAM by variables of different types (primitive ones like qubits, $n$-bit integers, tuples, vectors, etc.) holding quantum states
  - can be realized by translating the control flow and variable assignments and usages in Silq programs into quantum circuits
Comparison QisKit versus Silq - Deutsch-Jozsa Algorithm

Qiskit:

```python
def dj_algorithm(oracle, n):
    dj_circuit = QuantumCircuit(n+1, n)
    dj_circuit.x(n)
    dj_circuit.h(n)
    for qubit in range(n):
        dj_circuit.h(qubit)
        dj_circuit.append(oracle, range(n+1))
    for qubit in range(n):
        dj_circuit.h(qubit)
    for i in range(n):
        dj_circuit.measure(i, i)
    return dj_circuit
```

No need of uncomputation
⇒ Similar lengths of Qiskit and Silq codes!

Silq:

```python
def deutsch_jozsa[n::N]
    (f: const int[n]! → lifted 1:
    cand := 0:int[n];
    for k in [0..n]:
        cand[k] := H(cand[k]);
    target := H(1: 1);
    if f(cand) {
        target := X(target);
    }
    for k in [0..n]:
        cand[k] := H(cand[k]);
    result := measure(cand);
    return result == 0;
```

Phase-flipping `X` gate is replaced by built-in `phase` function, which applies a global phase to the whole quantum state
⇒ No ancillar qubit needed
Missing Features

- Composite data types
- Support of other programming paradigms like object-oriented software development
- Modularity
- Ability to split the code into multiple files
- Visualizations
- Standard functions
- Running on real quantum computer
- Quantum indexing to retrieve and modify $n$-qubit integers
Summary and Conclusions

- Circuits versus Control Flow - Programs
- Static type checking prevents common quantum computing errors
- Deferred Measurement
- Manual uncomputation
- Automatic uncomputation via static type checking
- Easy to learn language constructs instead of complex circuits for features like
  - quantum indexing and
  - support of QRAM