Lecture

Quantum Computing

(CS5070)

Introduction to Silq

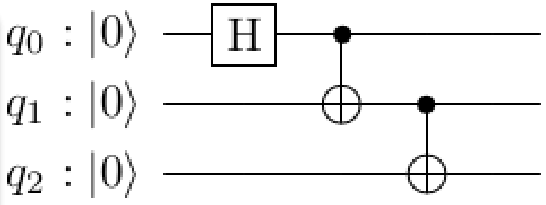
Professor Dr. rer. nat. habil. Sven Groppe
https://www.ifis.uni-luebeck.de/index.php?id=groppe

Motivation: Circuits versus Control Flow

- **Qiskit:** Developers have to think about how to solve their problems by realizing circuits
  - +/- familiar to physicists/electrical engineers

Example:

```
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
  - +/- 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

Example:

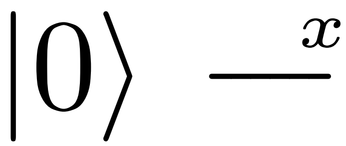
```
def ghz(){
 a:=0:𝔹;
 b:=0:𝔹;
 c:=0:𝔹;
```

```
 a:=H(a);
 if a { b := X(b); }
 if b { c := X(c); }
 return (a,b,c);
}
```

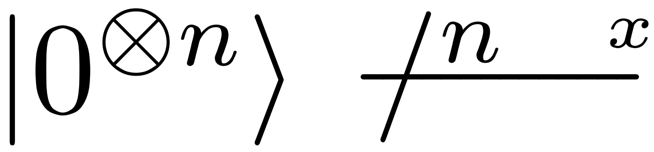
```
def main() {
 circuit = ghz();
}
```

Silq: Control Flow

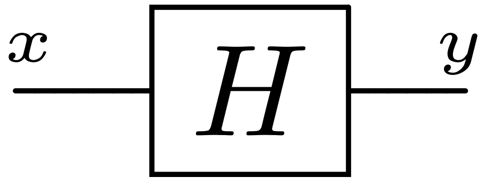
```
x:=0:𝔹;
```




```
x:=0:int[n];
```




```
y:=H(x);
```




```
x:=H(x);
```

```
if x {
  y:=H(y);
}
```

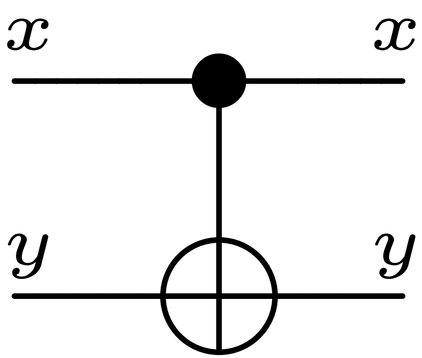
```
for k in [0..n) {
  x[k]:=H(x[k]);
}
```

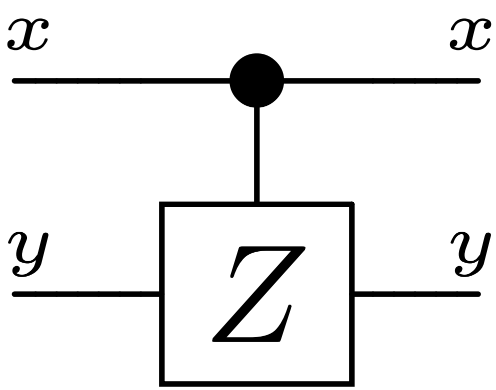
Silq: Multi-qubit quantum logic gates 1/2

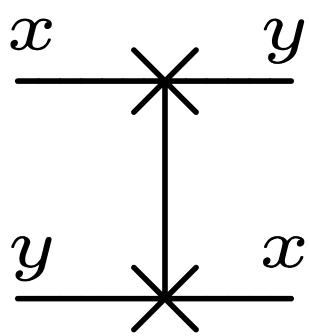
```
def CX(const x:𝔹,y:𝔹):𝔹{
 if x {
   y := X(y);
 }
 return y;
}
```

```
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);
}
```





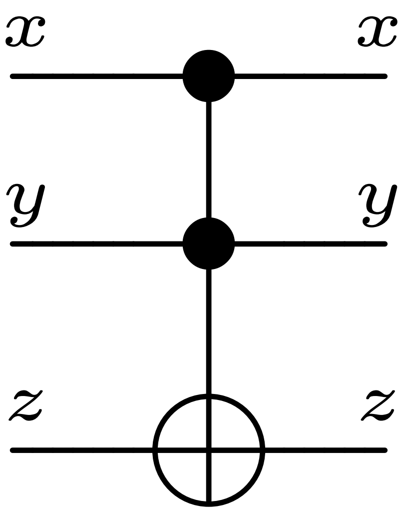


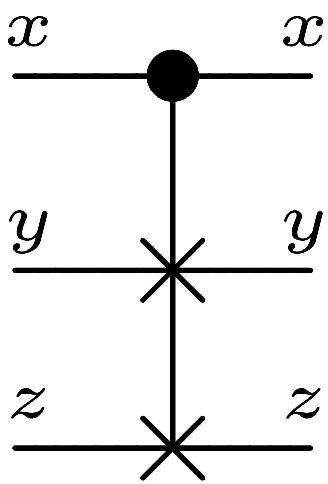
Silq: Multi-qubit quantum logic gates 2/2

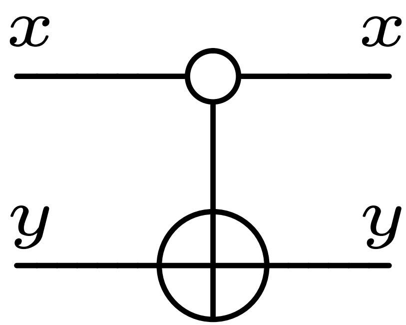
```
def CCX(const x:𝔹,
        const y:𝔹,z:𝔹):𝔹{
 if x && y {
   z := X(z);
 }
 return z;
}
```

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

```
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

Symbol	Alt. Symbol	Must be classical	Description
$\mathbb{1}$	1		The singleton type that only contains element `()`
`!τ`		result	type `τ`, but restricted to classical values
$\mathbb{B}$	B		Booleans, i.e. bits (!$\mathbb{B}$) or qubits ($\mathbb{B}$)
$\mathbb{N}$	N	✓	Natural numbers `0`, `1`, …
$\mathbb{Z}$	Z	✓	Integers …, `-1`, `0`, `1`, …
$\mathbb{Q}$	Q	✓	Rational numbers
$\mathbb{R}$	R	✓	Reals. Simulation semantics are implementation-defined (typically floating point)
`int[n]`			n-bit integers encoded in two's complement
`uint[n]`			`n`-bit unsigned integers
`τ×…×τ`	`τ x … x τ`		tuple types, e.g., $\mathbb{B}$`×int[n]`
`τ[]`			dynamic-length arrays
`τ^n`			vectors of length `n`
`[const] τ×…× [const] τ → [mfree|qfree] τ`			functions, optionally annotated as `mfree` or `qfree`, whose input types are optionally annotated as `const`

`n` stands for an arbitrary expression of type !$\mathbb{N}$

Annotations - Classical types (!)

- Duplicate classical annotations can be ignored: `!!τ ≡ !τ`
- Classical annotations commute with
  - tuples: `!(τ × … × τ) ≡ !τ × … × !τ`
    `⇒ !(τ × τ) ≡ !(τ × !τ) ≡ !(!τ × τ) ≡ !(!τ × !τ)`
  - arrays: `!τ[] ≡ (!τ)[] ≡ !(τ[])`
  - fixed-length arrays: `!τ^n ≡ (!τ)^n ≡ !(τ^n)`
- Classical values can be re-interpreted as quantum values:
  `!τ` ⊂ `τ` (with `τ` quantum type)
- No special issues to be considered for functions with only classical values (classical parameters, classical return value and classical variables):

```
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:

```
def myEval(const x:𝔹,f:const 𝔹!→𝔹){
  return f(x);
}
```

Annotations - `mfree`

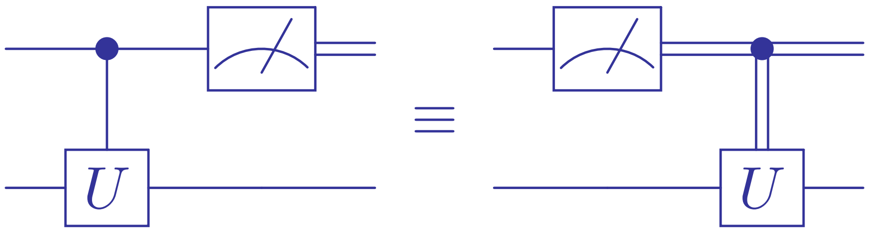
- Annotation mfree indicates that a function evaluation is guaranteed to be without applying any measurements

Example:

```
def myEval(f:𝔹→mfree 𝔹) mfree{
  return f(false); //   => myEval is mfree
}
```

Deferred Measurement Principle

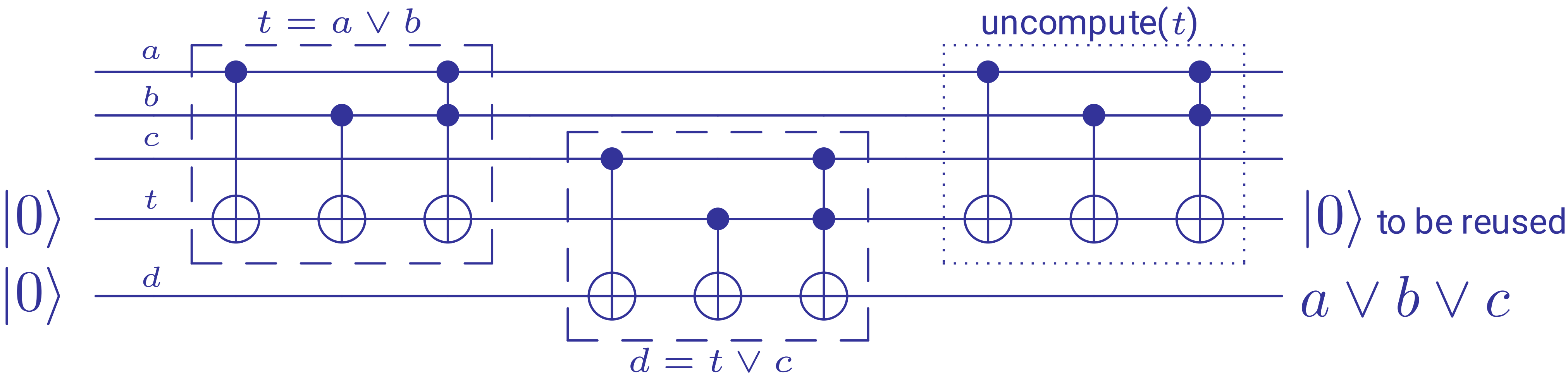
- Measuring commutes with conditioning:
  ⇒ The choice of whether to measure a qubit before, after, or during an operation conditioned on that qubit will have no observable effect on a circuit's final outcomes



  - 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 || b || c;
```

- Ancilla qubits must be explicitly uncomputed in other languages:

Qiskit:

```
c = QuantumCircuit(5)
c.cx(0,3);c.cx(1,3);c.ccx(0,1,3)
c.cx(2,4);c.cx(3,4);c.ccx(2,3,4)
c.cx(0,3);c.cx(1,3);c.ccx(0,1,3)
```

Quipper:

```
with_computed (OR a b) $
 \t -> 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) <- a11_FetchE tau_j ee eed
  cTri <- foldM (\cTri k -> do
    let tau_k = tau ! k
    eedd_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
  (taub,ee,eed) <- a11_FetchE tau_j ee eed
  qterm (intMap_replicate rr False) eed
  return cTri)
 cTri [0..rrbar-1]
```

QWire:

```
index  : ∏(n:Nat,i:Nat) . CIRC(t[n]        ,t[n]⊗        t)
     = ...
qindex : ∏(n:Nat,m:Nat) . CIRC(t[n]⊗qubit[m],t[n]⊗qubit[m]⊗t)
     = ...
controlledInc : ∏(n:Nat). CIRC(qubit[n]⊗qubit,qubit[n]⊗qubit)
     = ...

EvalCondition : ∏(r:Nat,rrbar:Nat,j:Nat,k:Nat). CIRC(
  qubit[rrbar][rrbar]⊗qubit[rrbar][r]⊗qubit[rrbar],...⊗qubit
) = box(ee,tau,eew) =>
  (tau,tauj) <- unbox (index rrbar j) tau; -- tauj=tau[j]
  (tau,tauk) <- unbox (index rrbar k) tau; -- tauk=tau[k]
  (ee, tauj, eed) <- unbox (qindex rrbar r) ee tauj; -- eed=ee
      [tauj]
  (eed,tauk,eedd_k) <- unbox (qindex rrbar r) eed tauk; --
      eedk=eed[tauk]
  (eew,eewj) <- unbox (index rrbar j) eew; -- eewj=eew[j]
  (eew,eewk) <- unbox (index rrbar k) eew; -- eewk=eew[k]
  (eedd_k,eewj,eewk,c) <- unbox and eedd_k eewj eewk; --
      condition
  output (ee,tau,eew,tauj,tauk,eed,eedd_k,eewj,eewk,c) --
      output

LoopBody : ∏(r:Nat,rrbar:Nat,j:Nat,k:Nat). CIRC(
  qubit[rrbar][rrbar]⊗qubit[rrbar][r]⊗qubit[rrbar]⊗qubit[rrbar
      ],
  qubit[rrbar][rrbar]⊗qubit[rrbar][r]⊗qubit[rrbar]⊗qubit[rrbar
      ]
) = box (ee,tau,eew,cTri) =>
  (ee,tau,eew,tauj,tauk,eed,eedd_k,eewj,eewk,c) <- unbox (
      EvalCondition r rrbar j k) ee tau eew; -- evaluate
      condition
  (cTri,c) <- unbox (controlledInc rrbar) cTri c; --
      controlled increment
  (ee,tau,eew) <- unbox (reverseIsometric EvalCondition r
      rrbar j k) ee tau eew tauj tauk eed eedd_k eewj eewk c
      -- uncompute
  output (ee,tau,eew,cTri) -- output
```

Automatic Uncomputation & Concise Language Constructs reduce Code Size

Q#

Silq

-44%
code

-50%
library

- Comparison on Microsoft's Q# coding contests, see 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.

Function	Signature	qfree	Remarks
`H`	`𝔹 → mfree 𝔹`		`H` introduces superpositions
`X`	`𝔹 → qfree 𝔹`	✓	`X` neither introduces nor destroys superpositions
`x || y`	`const 𝔹 × const 𝔹 → qfree 𝔹`	✓	Logical disjunction neither introduces nor destroys superpositions
`def myEval(f:𝔹→qfree 𝔹) qfree{ return f(false); }`	`(𝔹 → qfree 𝔹) → qfree 𝔹`	✓	`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:

```
def MyOr(x:𝔹, y:!𝔹)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)

    $\Rightarrow$ trying to forget variable, trying to uncompute variable
    - If automatic uncomputation is not possible and manual forgetting is not specified $\Rightarrow$ implicit measurement with side-effects $\Rightarrow$ Silq rejects code

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

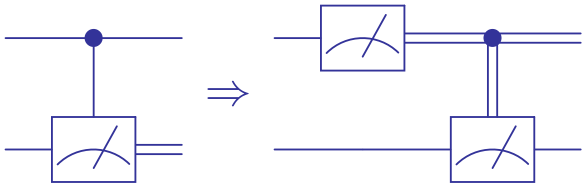
- Declaring parameter `x` to be `const` $\Rightarrow$ `x` has to be consumed or uncomputed by callee of the function

```
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

```
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:

```
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:

```
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

```
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:

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

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

- `forget(x=y)` is (unsafe) shorthand for `reverse(dup[𝔹])(x,y)`

Using Consumed Variable

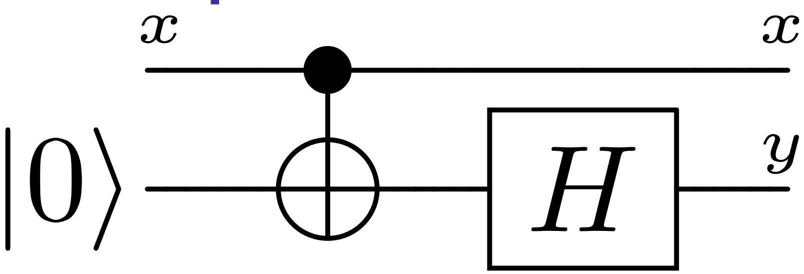
- **Error**: Accessing already consumed variables (which are no longer available after being consumed)

```
def useConsumed(x:𝔹){
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)

```
def duplicateConst(const x:𝔹){
y := H(x);
return (x,y); // no error
}
```

Silent **duplication** of constant x:



- All duplicates of constant variables are either consumed (as above), or can be uncomputed

Impossible Uncomputation - Not Constant

```
def nonConst(y:𝔹){
  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`

```
def signFlipOf0(const y:𝔹){
  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`

```
def nonQfree(const y:𝔹,z:𝔹){
  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

- `e1[e2]` for non-classical `e2`, if `e1` 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:
  $W_{2^k} = \frac{|100...0\rangle + |010...0\rangle + ... + |00...01\rangle}{\sqrt{2^k}}$:

```
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!
  // 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

```
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:

```
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:

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

```
def deutsch_jozsa[n:!ℕ]
   (f: const int[n]!
       → lifted 𝔹):!𝔹{
 cand := 0:int[n];
 for k in [0..n){
   cand[k] := H(cand[k]);
 }
 if f(cand) {
    phase(π);
 }
 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