Programming language for unified computing with classical and quantum bits

Gergely Gálfi, PhD Student, Eötvös University
Dr. Tamás Kozsik, Eötvös University
Dr. Zoltán Zimborás, Wigner Institute

Motivation - selfish

- Previous project on simulation of quantum computers with Markov Chain Monte Carlo method
  - Not to be discussed in detail, just to provide a possible motivation
- MCMC-based quantum methods rely on a good “small step” strategy for walking around the phase space (incoming and outgoing states of each individual step)
- For simple quantum algorithms it is easy to find a proper small step strategy
  - QFT: the small step could be the increasing or decreasing of the in-out states as binary n-bit numbers
- Challenge: to develop and test more general small step strategies which can work on less symmetrical algorithms (e.g. LCGs, hash-coders, etc.)

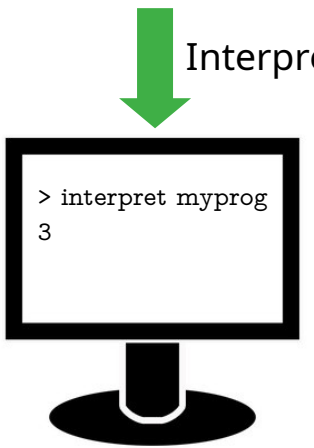
Motivation – not so selfish

- Many of the quantum algorithms have steps which are quantized version of some common classical procedures.
  - Shor's algorithm – consecutive controlled modular multiplications:
    - $x = y*a^k \bmod N$, where: a, k, N are classical; x , y are qubit-based integers.
- It is desirable to have a development framework which allows to define classical-like algorithms, and a "quantum-executable" could be generated out of it.
- One possible solution for that tool set is a programming language which is flexible enough to blend classical and quantum bits
- Qubla (QUBit LAnguage)

Interpreter vs. compiler

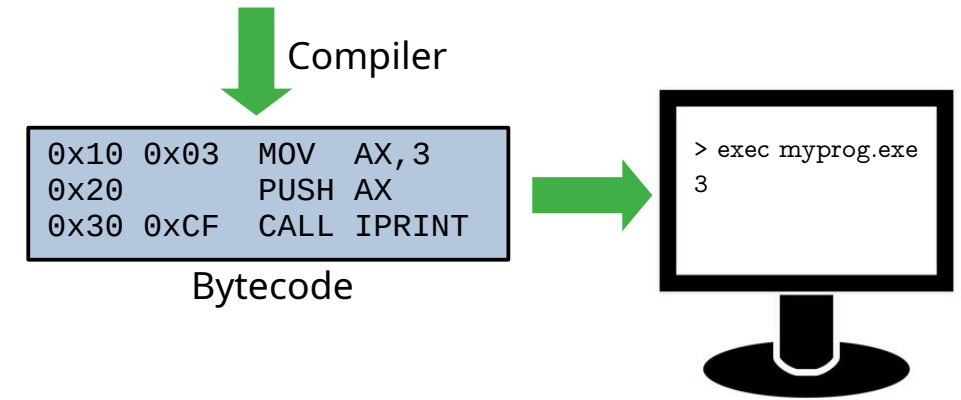
Interpreter

```
x=3;
print(x);
```

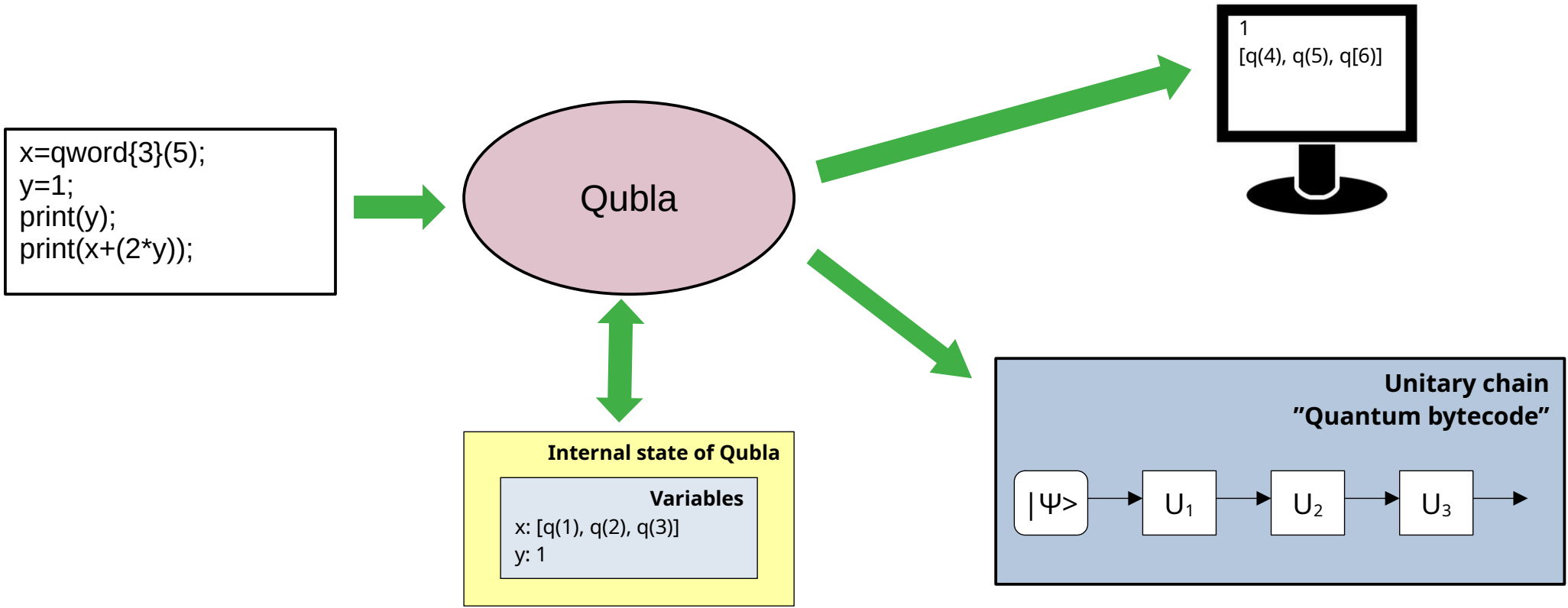


Compiler

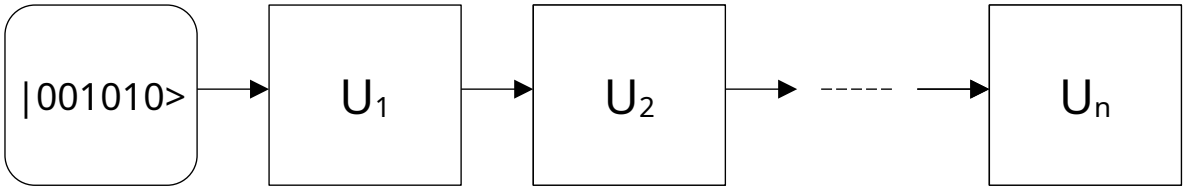
```
x=3;
print(x);
```

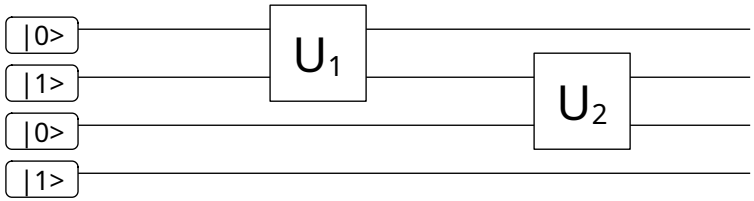


Qubla as an interpreter-compiler hybrid



Unitary chain





$$(U_2 |\Psi\rangle)_{i1,i2,i3,i4;\ k1,k2,k3,k4} = \Sigma_{p,q}\ U_{2;\ i2,i3;\ r,s}\ \Psi_{i1,r,s,i4}$$

Elementary unitary operators in all practical cases act on maximum 6 qubits, so they could be represented by a 64 X 64 complex matrix.

Unitary transformations in Qubla

- Special class of unitary transformations: when they implements boolean functions
- Formally: for an f: $B^N \rightarrow B^N$, we say U implements f iff
  $U|x_1...x_N\rangle = |f(x)_1...f(x)_N\rangle$
- Could be generalized to any f: $B^m \rightarrow B^n$, it requires padding, like
  $U|x_1...x_m\rangle = |f(x)_1...f(x)_n\, 0...0\rangle$
- With the padding above, U is unitary iff f is injective
- More general unitary operators which doesn't originate from a Boolean function are allowed, and used as given.

What Qubla is not

- What was **not** amongst the goals: to simulate/run the quantum system. What we expect from the Qubla compiler is only to generate an unambiguous quantum logic which could be a basis for a further simulation or an implementation on a real-world quantum hardware.
- Our main focus was on generating and optimizing the unitary part of quantum algorithms
- That comes with the restriction that no non-unitary steps are allowed
- Measurements should be done outside of Qubla

Qubla - Problem of invariance

- Example code:
```
x = 5;
y = x + 1;
z = x – 1;
```
- In a classical programming languages it is usually guaranteed that variables involved only in read-only operations doesn't change their values (in the example `z == 4` after last statement)
- Quantum operations are generally affects **all** the qubits spanning the state of the system.
- Definition: We say that a qubit is *invariant* under a series of quantum steps, if in the case these steps applied on any pure canonical base state (like |0101>), then the density matrix of the qubit doesn't change.
- The compiler guarantees the invariance till only Boolean-originated transformations are applied on the given qubits. Steps of Non-Boolean origin can "harm" – usually that is what we want in those cases (e.g. QFT)
- Invariance is achieved by adding all the input qubits to the outputs.
- This strategy is very greedy on qubits – later some unused could be removed

Qubla - Syntax

Function defined by truth table:

```
function add3bit table {
    [0, 0, 0] : [0, 0],
    [1, 0, 0] : [1, 0],
    [0, 1, 0] : [1, 0],
    [1, 1, 0] : [0, 1],
    [0, 0, 1] : [1, 0],
    [1, 0, 1] : [0, 1],
    [0, 1, 1] : [0, 1],
    [1, 1, 1] : [1, 1]
}
```

Function defined by script:

```
function `+`(x, y){
    local n = len(x)
    local ret=alloc(n), carry = bit0, addout;
    for(i : seq(n)){
        addout = add3bit(x[i], y[i], carry);
        ret[i] = addout[0];
        carry = addout[1];
    }
    return word(ret);
}
```

Qubla – compilation and execution of truth tables

- Example: `add3bit(1, qbit[5], qbit[6])`
- To guarantee the invariance of the input qubits (`qbit[5]` and `qbit[6]`), they should be (temporarily) added to the outputs – we omit this for sake of simplicity

Qubla – compilation and execution of truth tables

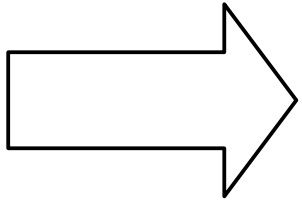
bit(1)	qbit[7]	qbit[8]	sum	carry
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Qubla – compilation and execution of truth tables

bit(1)	qbit[7]	qbit[8]	sum	carry
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Qubla – compilation and execution of truth tables

bit(1)	qbit[7]	qbit[8]	sum	carry
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1



First the compiler produces a truth table to be applied only on qubits

qbit[7]	qbit[8]	sum	carry
0	0	1	0
1	0	0	1
0	1	0	1
1	1	1	1

Qubla – compilation and execution of truth tables

qbit[7]	qbit[8]	sum	carry
0	0	1	0
1	0	0	1
0	1	0	1
1	1	1	1

Further simplifications in certain cases (not for this example):

- Constant output column could be handled as classical instead of qubits
- Searching of input columns within the outputs
  - If the compiler founds such, it removes the preliminary copy step of that qubit

Qubla – compilation and execution of truth tables

qbit[7]	qbit[8]	sum	carry
0	0	1	0
1	0	0	1
0	1	0	1
1	1	1	1

Problem:

- Quantum transformations ought to be unitary
- A boolean function is a unitary transformation iff it’s output lines are permutation of input lines

Qubla – compilation and execution of truth tables

qbit[7]	qbit[8]	sum	carry
0	0	1	0
1	0	0	1
0	1	0	1
1	1	1	1

Problem:

- Quantum transformations ought to be unitary
- A boolean function leads to a unitary transformation iff it is injective (one-to-one, invertible, etc.)
- This condition is generally not held (neither in our example)

Qubla – compilation and execution of truth tables

qbit[7]	qbit[8]	sum	carry
0	0	1	0
1	0	0	1
0	1	0	1
1	1	1	1

- We can achieve injectivity if we add carefully chosen group-splitting columns
- If $\max(n_g)$ is the size of the largest group with same output
- $\log_2(\max(n_g))$ new columns are needed, but could be chosen to be enough as well

Qubla – compilation and execution of truth tables

qbit[7]	qbit[8]	sum	carry	qbit[9]
0	0	1	0	0
1	0	0	1	0
0	1	0	1	1
1	1	1	1	0

- We can achieve injectivity if we add carefully chosen group-splitting columns
- If $\max(n_g)$ is the size of the largest group with same output
- $\log_2(\max(n_g))$ new columns are needed, but could be chosen to be enough as well

Qubla – compilation and execution of truth tables

qbit[7]	qbit[8]	qbit[7]	qbit[8]	qbit[9]
0	0	1	0	0
1	0	0	1	0
0	1	0	1	1
1	1	1	1	0

- Finally qubits are assigned to each output bit.
- It could happen that there are more output than input bits → new qubits are initialized (in our example: qbit[9]).
- The tricks above only provide injectivity, but by extending these functions bijectivity also easily achieved.
- The extension is not unique, and it is not done by the compiler as this ambiguity could be used as a freedom in the hardware implementation.

Qubla – compilation and execution of truth tables

qbit[7]	qbit[8]	qbit[7]	qbit[8]	qbit[9]
0	0	1	0	0
1	0	0	1	0
0	1	0	1	1
1	1	1	1	0

Summary:

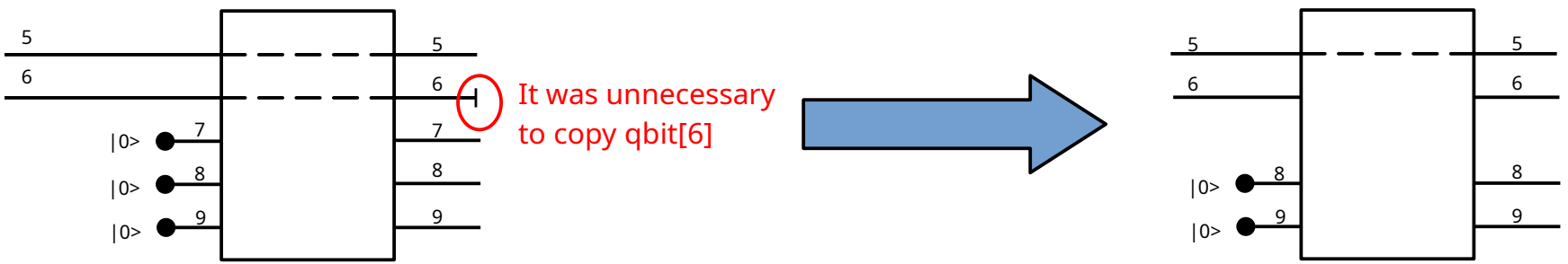
- Original function call:
  `add3bit(1, qbit[7], qbit[8])`
- Returned value of the function call:
  `[qbit[7], qbit[8]]`

Reduction of quantum chains

- It is apparent that Qubla compiler is greedy on qubits and quantum transformations
- Some reduction of raw quantum logic is needed
- Different needs for different purposes
  - A real hardware implementation needs both the reduction of the number of (logical) qubits and the number of the steps
  - Simulation is sensitive on the number of the qubits only → we focused on this, so some important reduction methods (like uncomputation) are not yet addressed
- In every Qubla code the qubits which we intend to use (to measure, to feed into another quantum algorithm, ...) should be explicitely specified (with the help of the `output` function)
- Going from the last step backward we try to remove all the steps which neither directly nor implicitely impacting the output qubits (in the classical sense)

Reduction of quantum chains

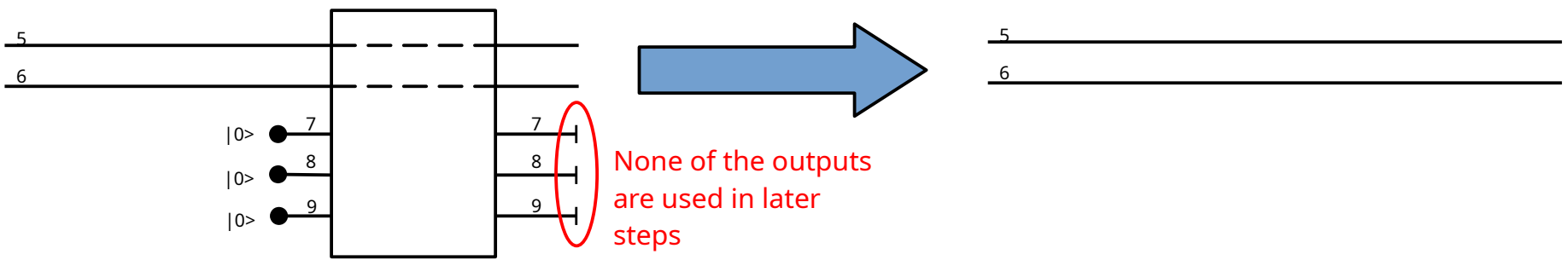
When there is a step where some input qubits were “protected” by copying to the output, but one of them isn’t used in any later step or isn’t declared as script output, then the corresponding replication step is removed and the copy of the qubit is substituted by the original one



Reduction of quantum chains

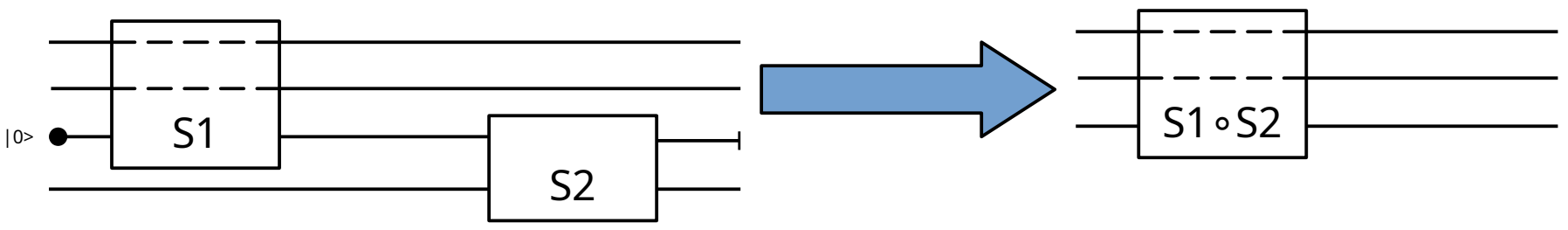
When there is a step where none of the output qubits are used in later steps or declared as script output, then the whole step is removed

- This situation could happen realistically when a truth table is fed with mixed quantum-classical bits, and not all outputs are used in the code (e.g. carry bits)



Step joining

- It frequently happens that a qubit is only created in a step to be consumed by another step later in the process
- It is desirable to join or “compose” these step to get rid of intermediate qubits

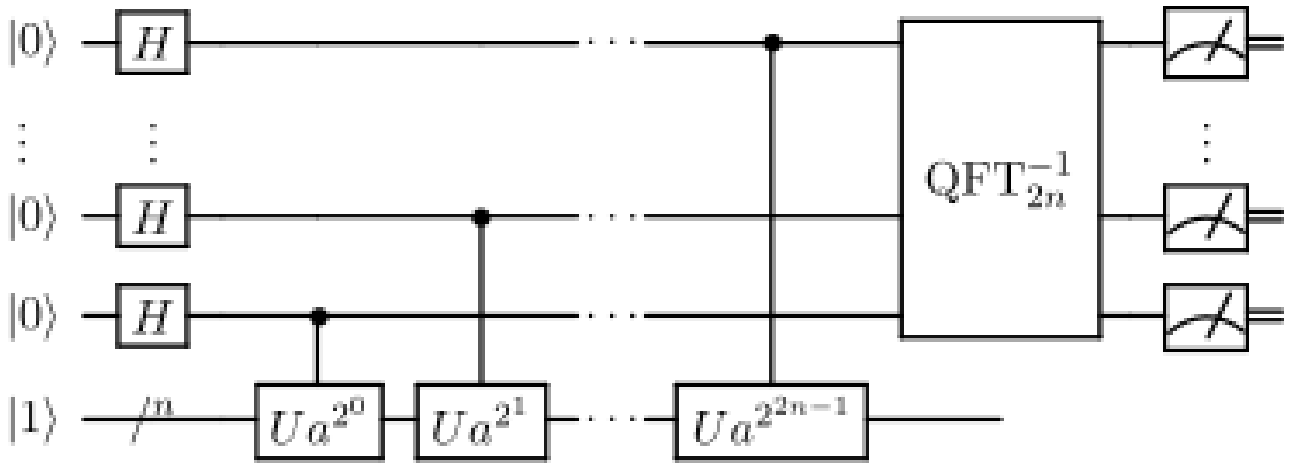


Remedy – Shor's algorithm

- Integer factorization on Quantum Computers
- N is a (large) integer, find integers p, q where N = p*q
- Pick a random integer 2 <= a < N
- Find the shortest period of $f(x) = a^x \bmod N$, i.e. $f(x + r) = f(x)$
- $p = \gcd(a^{r/2} - 1, N)$ is a good factor

Remedy – Shor's algorithm

- Find the shortest period of $f(x) = a^x \bmod N$

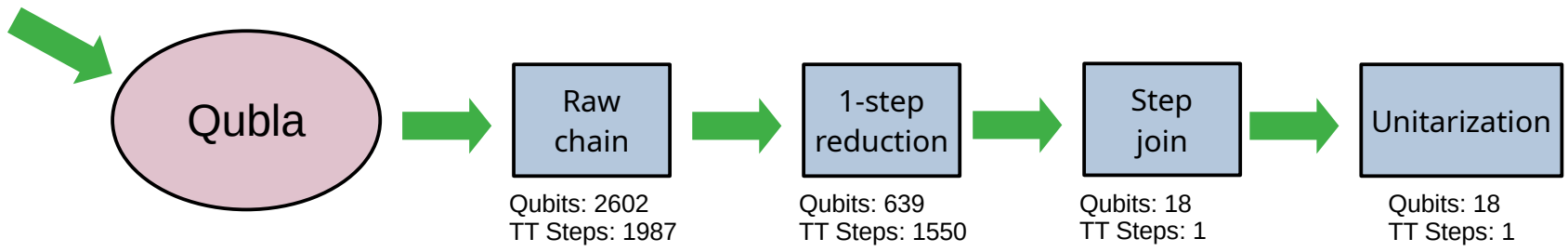


Shor algorithm in Qubla

N=35; n_{bits}=6; a=17

```
import qft;
import modular;

n = 6;
N = 35;
a = 17;
H = {0 : 1 / sqrt(2), 1 : 1 / sqrt(2)};
lstr = alloc(2*n);
for(i : seq(2*n))
    lstr[i] = qstate(H)[0];
r = quword(lstr);
x = modexp(uword{n}(a), r, N);
fr = qft(r);
output(x);
output(fr);
```



Further steps

- Last building block: develop automatic procedures to transform the resulting quantum logic into physically realizable quantum circuits
  - Interfacing with Qiskit or other quantum cloud API
  - There is a brute force decomposition for Boolean-origin unitaries – extremly costly in CNOT count & supplementary qubits
  - Current alternative: Squander (Rakyta et al.)
  - A more discrete, permutation matrix-oriented decomposition is wanted

Thank you for your attention!

github.com/ggalfi/qubla