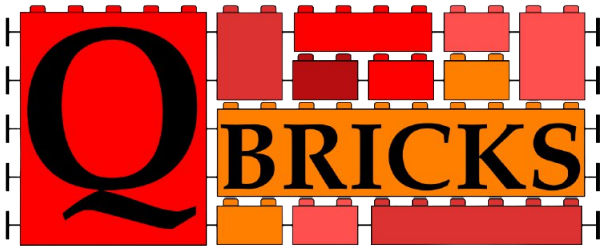
cea list

Quantum programming and formal verification (Qbricks)



Christophe Chareton, Mathieu Nguyen, Sébastien Bardin

Quantum programming and formal verification (Qbricks)

1. **Specifications and programs requirements in quantum computing**
2. **Hybrid symbolic execution**
3. **Unitary formal specification proof**

Quantum memory states

Potential for exponential acceleration !

- Classical world:

One sequence in {🐱, ✗}^{n} (over 2^n possible)

- Quantum world:

α_0 🐱$_0$ 🐱$_1$ 🐱$_2$ 🐱$_3$ ... 🐱$_{n-1}$

α_1 🐱$_0$ 🐱$_1$ 🐱$_2$ 🐱$_3$ ... ✗$_{n-1}$

...

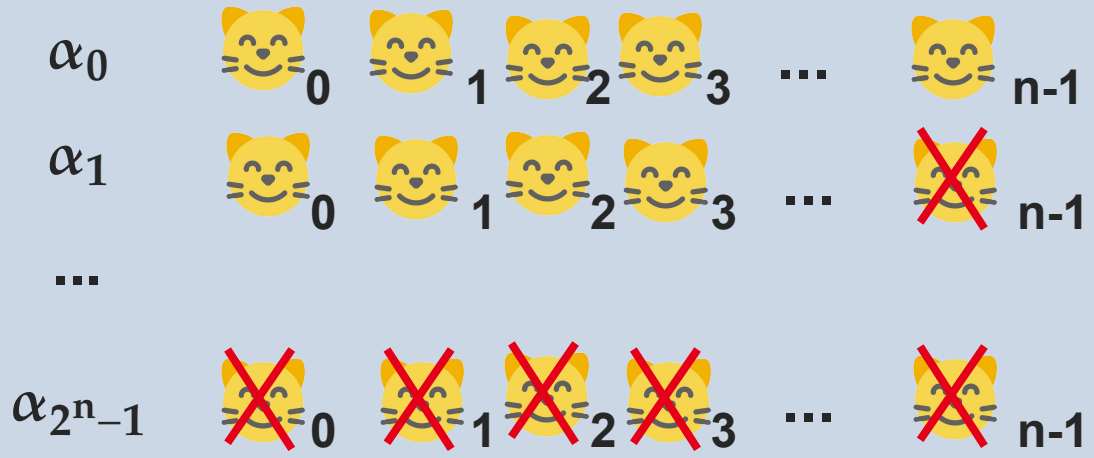
α_{2^n-1} ✗$_0$ ✗$_1$ ✗$_2$ ✗$_3$ ... ✗$_{n-1}$

Quantum memory states

+ Some *strange* rules :

-
  - Restricted set of operations : **unitarity**
  - **Destructive + prôbabilistic measurement**

- Quantum world:



The hybrid model

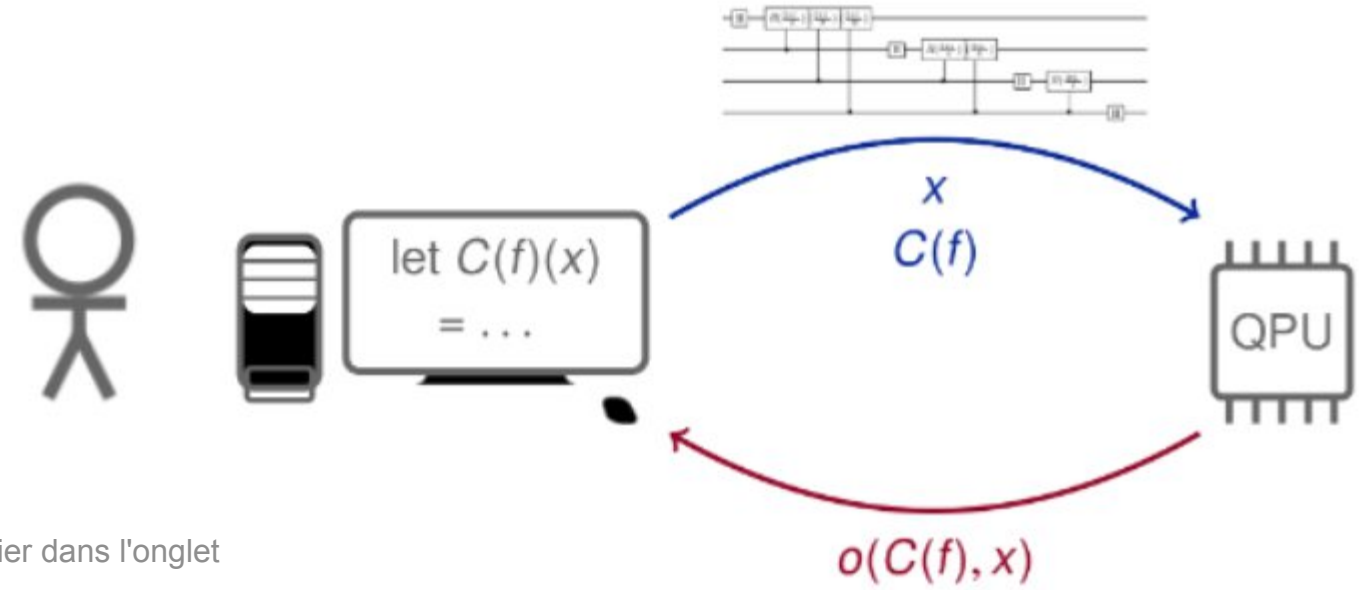
A quantum co-processor (QPU), controlled by a classical computer

- classical control flow
- CPU ⇒ QPU : quantum computing requests, sent to the QPU

→structured sequenced of instructions: **quantum circuits**

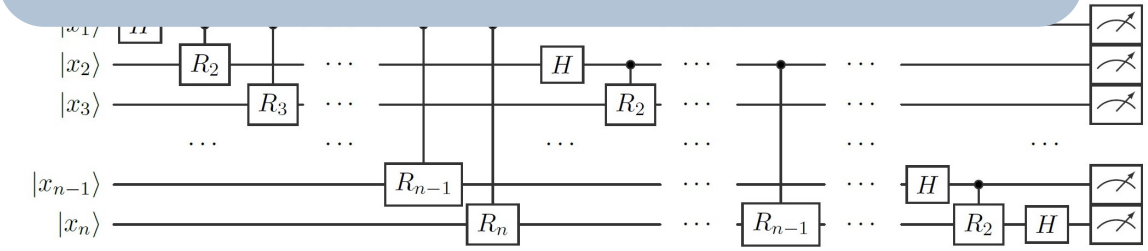
- initialise quantum data
- run pure quantum physics operations
- non-deteministic measure : quantum data —> classical data

- QPU ⇒ CPU: **probabilistic** computation results (**classical** information)

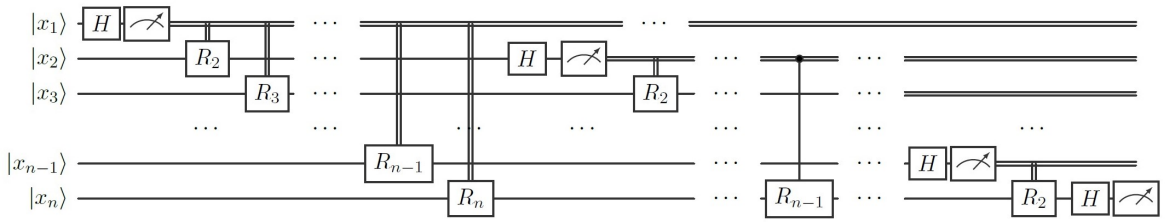


The hybrid model : In practice, embedding runtime control

Hybridization of quantum primitives

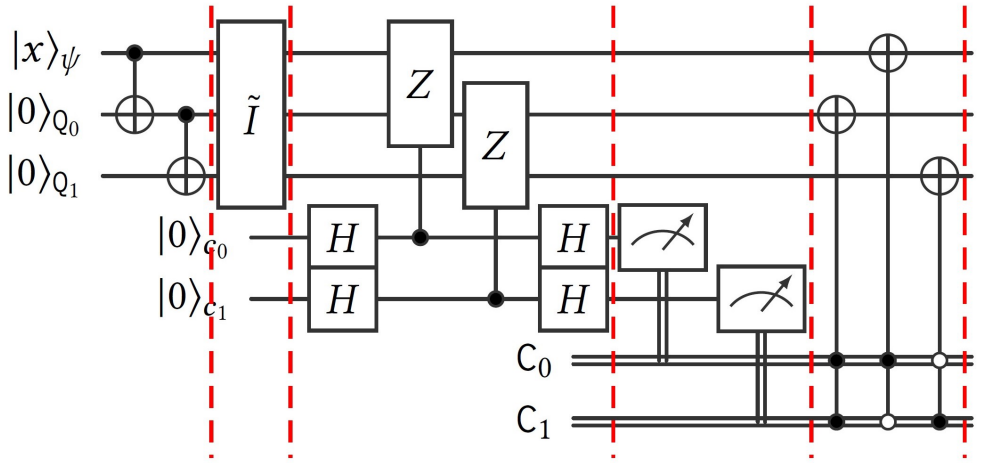


(a) The standard QFT unitary circuit



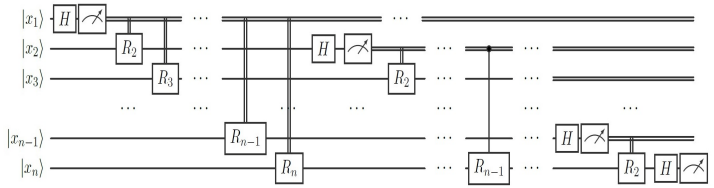
(b) A dynamic version of the QFT

Error correction

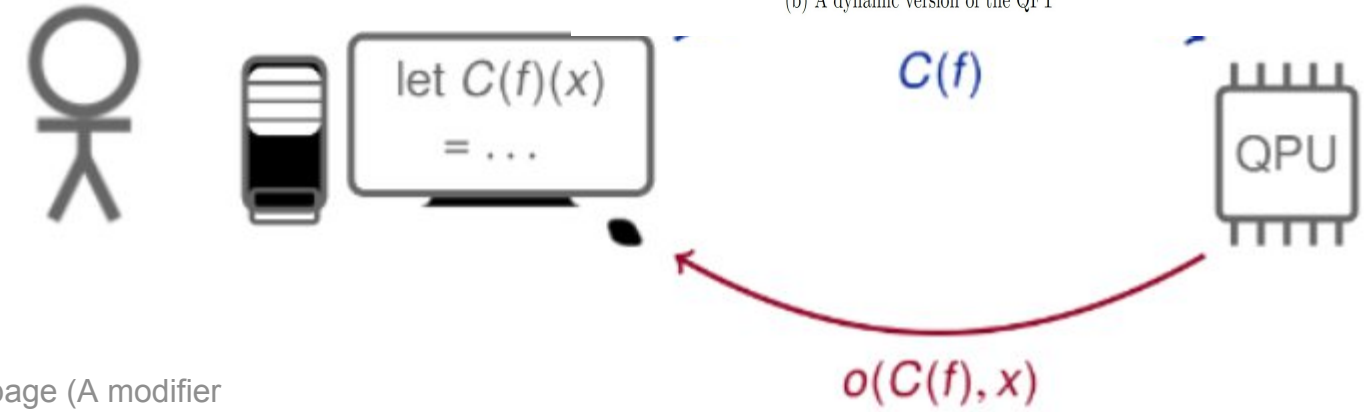


The hybrid model

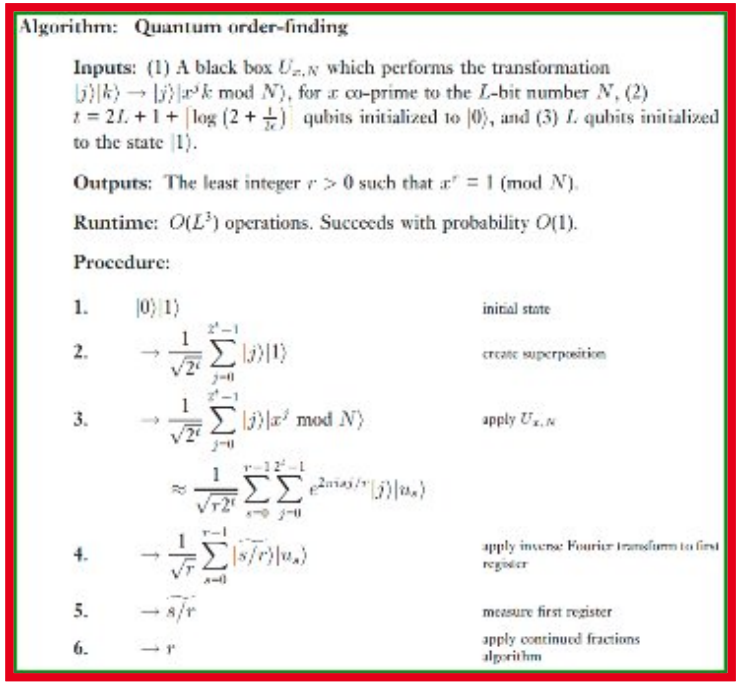
- « QPU » must support classical data runtime control
- Embedded classical computation instruction
- Discriminate central unit/QPU instructions
- Different compilation requirements



(b) A dynamic version of the QFT



Verification : specifications



Algorithm: Quantum order-finding

Inputs: (1) A black box $U_{x,N}$ which performs the transformation $|j\rangle|k\rangle \rightarrow |j\rangle|x^j k \bmod N\rangle$, for x co-prime to the L-bit number N, (2) $t = 2L + 1 + \lceil \log(2 + \frac{1}{2\epsilon}) \rceil$ qubits initialized to $|0\rangle$, and (3) L qubits initialized to the state $|1\rangle$.

Outputs: The least integer $r > 0$ such that $x^r = 1 \pmod N$.

Runtime: $O(L^3)$ operations. Succeeds with probability $O(1)$.

Procedure:

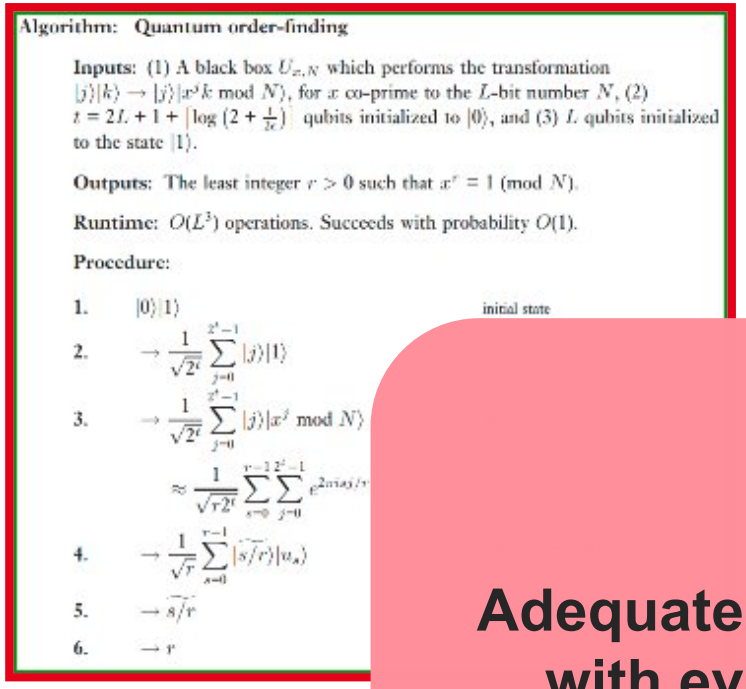
1. $|0\rangle|1\rangle$ — initial state
2. $\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|1\rangle$ — create superposition
3. $\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|x^j \bmod N\rangle$ — apply $U_{x,N}$

   $\approx \frac{1}{\sqrt{r2^t}} \sum_{s=0}^{r-1} \sum_{j=0}^{2^t-1} e^{2\pi i s j / r} |j\rangle|u_s\rangle$
4. $\rightarrow \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |\widetilde{s/r}\rangle|u_s\rangle$ — apply inverse Fourier transform to first register
5. $\rightarrow \widetilde{s/r}$ — measure first register
6. $\rightarrow r$ — apply continued fractions algorithm

A specification preamble:

- **Input parameters (size, oracle, etc)**
- **Functional correctness**: Inputs-Outputs relation
- **Complexity**: number of elementary operations

Verification : specifications



Algorithm: Quantum order-finding

Inputs: (1) A black box $U_{x,N}$ which performs the transformation $|j\rangle|k\rangle \rightarrow |j\rangle|x^j k \bmod N\rangle$, for x co-prime to the L-bit number N, (2) $t = 2L + 1 + \lceil \log(2 + \frac{1}{2\epsilon}) \rceil$ qubits initialized to $|0\rangle$, and (3) L qubits initialized to the state $|1\rangle$.

Outputs: The least integer $r > 0$ such that $x^r = 1 \pmod N$.

Runtime: $O(L^3)$ operations. Succeeds with probability $O(1)$.

Procedure:

1. $|0\rangle|1\rangle$ initial state
2. $\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|1\rangle$
3. $\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|x^j \bmod N\rangle$
   $\approx \frac{1}{\sqrt{r2^t}} \sum_{s=0}^{r-1} \sum_{j=0}^{2^t-1} e^{2\pi i s j / r}$
4. $\rightarrow \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |\widetilde{s/r}\rangle|u_s\rangle$
5. $\rightarrow \widetilde{s/r}$
6. $\rightarrow r$

A specification preamble:

- **Input parameters (size, oracle, etc)**
- **Functional correctness**: Inputs-Outputs relation
- **Complexity**: number of elementary operations

Adequate implementation should come with evidence regarding the specs

Verification : specifications

Algorithm: Quantum order-finding

Inputs: (1) A black box $U_{x,N}$ which performs the transformation $|j\rangle|k\rangle \to |j\rangle|x^j k \bmod N\rangle$, for x co-prime to the L-bit number N, (2) $t = 2L + 1 + \lceil \log\left(2 + \frac{1}{2\epsilon}\right) \rceil$ qubits initialized to $|0\rangle$, and (3) L qubits initialized to the state $|1\rangle$.

Outputs: The least integer $r > 0$ such that $x^r = 1 \pmod N$.

Runtime: $O(L^3)$ operations. Succeeds with probability $O(1)$.

Procedure:

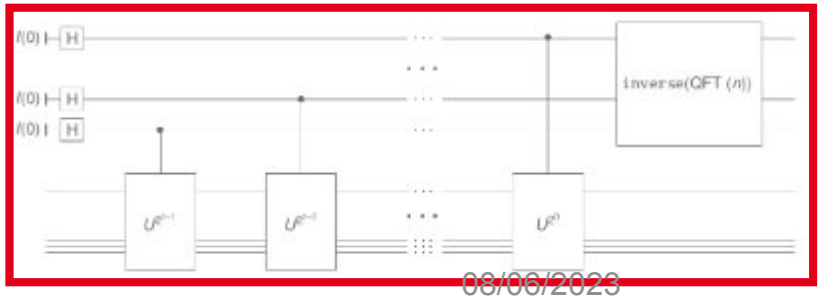
1. $|0\rangle|1\rangle$ — initial state
2. $\to \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|1\rangle$ — create superposition
3. $\to \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|x^j \bmod N\rangle$ — apply $U_{x,N}$

   $\approx \frac{1}{\sqrt{r2^t}} \sum_{s=0}^{r-1} \sum_{j=0}^{2^t-1} e^{2\pi i s j / r} |j\rangle|u_s\rangle$
4. $\to \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |\widetilde{s/r}\rangle|u_s\rangle$ — apply inverse Fourier transform to register
5. $\to \widetilde{s/r}$ — measure first register
6. $\to r$ — apply continued fractions algorithm

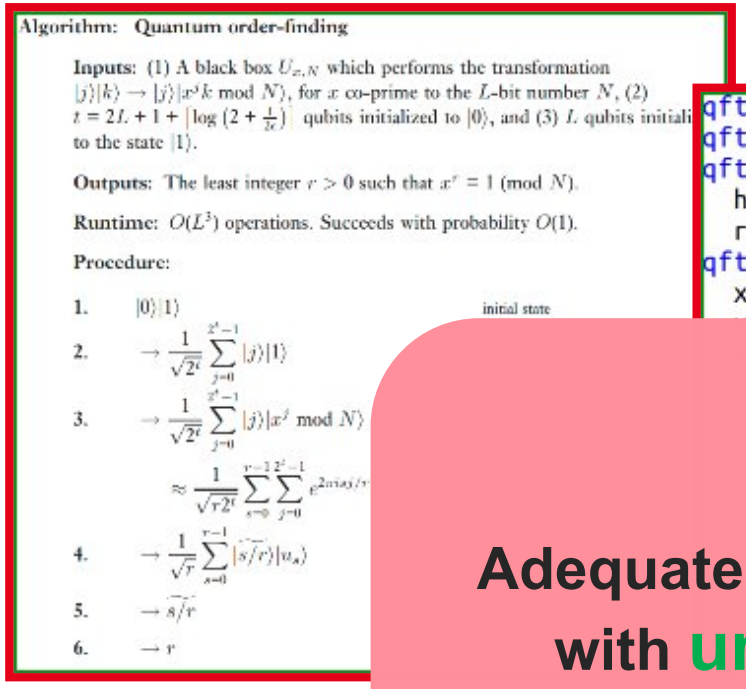
```
qft_internal :: [Qubit] -> Circ [Qubit]
qft_internal [] = return []
qft_internal [x] = do
  hadamard x
  return [x]
qft_internal (x:xs) = do
  xs' <- qft_internal xs
  xs'' <- rotations x xs' (length xs')
  x' <- hadamard x
  return (x':xs'')
  where
    -- Auxiliary function used by 'qft'.
    rotations :: Qubit -> [Qubit] -> Int -> Circ [Qubit]
    rotations _ [] _ = return []
    rotations c (q:qs) n = do
      qs' <- rotations c qs n
      q' <- rGate ((n + 1) - length qs) q `controlled` c
      return (q':qs')
```

A specification preamble:

- **Input parameters (size, oracle, etc)**
- **Functional correctness**: Inputs-Outputs relation
- **Complexity**: number of elementary operations



Verification : specifications



Algorithm: Quantum order-finding

Inputs: (1) A black box $U_{x,N}$ which performs the transformation $|j\rangle|k\rangle \rightarrow |j\rangle|x^j k \bmod N\rangle$, for x co-prime to the L-bit number N, (2) $t = 2L + 1 + \lceil \log(2 + \frac{1}{2\epsilon}) \rceil$ qubits initialized to $|0\rangle$, and (3) L qubits initialized to the state $|1\rangle$.

Outputs: The least integer $r > 0$ such that $x^r = 1 \pmod N$.

Runtime: $O(L^3)$ operations. Succeeds with probability $O(1)$.

Procedure:

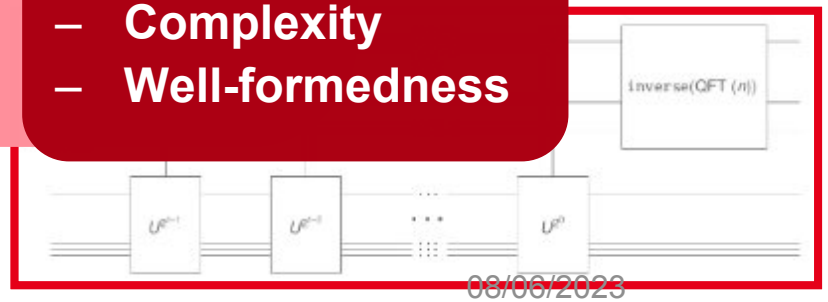
1. $|0\rangle|1\rangle$ initial state
2. $\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|1\rangle$
3. $\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|x^j \bmod N\rangle$
   $\approx \frac{1}{\sqrt{r2^t}} \sum_{s=0}^{r-1} \sum_{j=0}^{2^t-1} e^{2\pi i s j / r}$
4. $\rightarrow \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |\widetilde{s/r}\rangle|u_s\rangle$
5. $\rightarrow \widetilde{s/r}$
6. $\rightarrow r$

```
qft_internal :: [Qubit] -> Circ [Qubit]
qft_internal [] = return []
qft_internal [x] = do
  hadamard x
  return [x]
qft_internal (x:xs) = do
  xs' <- qft_internal xs
```

Adequate implementation should come with universally valid evidence regarding the specs

- **Functionality**
- **Complexity**
- **Well-formedness**

A specification
- **Input parame**
- **Functional co**
- **Complexity**: number of elementary operations



Standard debuguing techniques fail…

Potential method	Drawback
Assertion checking ?	Requires (destructive) **measurement** with highly superposed states
Final test ?	How to pinpoint **error source** ?
Simulation ?	As far as we don't need a Quantum Computer !

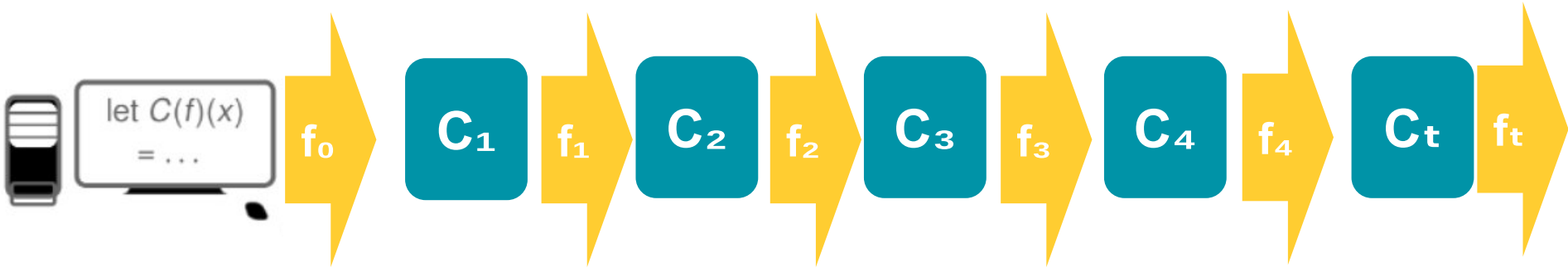
Standard debuguing techniques fail…
… the alternative of formal verification

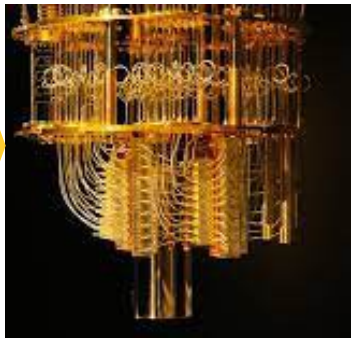
Potential method	Drawback
Assertion checking ?	Requires (destructive) **measurement** with highly superposed states
Final test ?	How to pinpoint **error source** ?
Simulation ?	As far as we don't need a Quantum Computer !

Testing/Assertion checking	Static analysis
executions/simulations	**static** analysis, **no need to execute**
b**ounded** parameters	**scale insensitive/any instance**
statistical arguments	absolute, **mathematical guarantee**

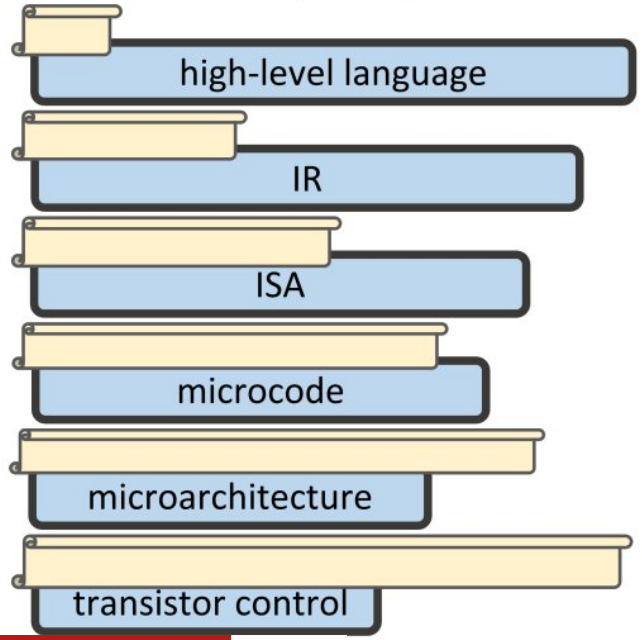
Build on **best practice** of formal verification for the classical case and **tailor them to the quantum** case

Guaranteeing the compilation stack

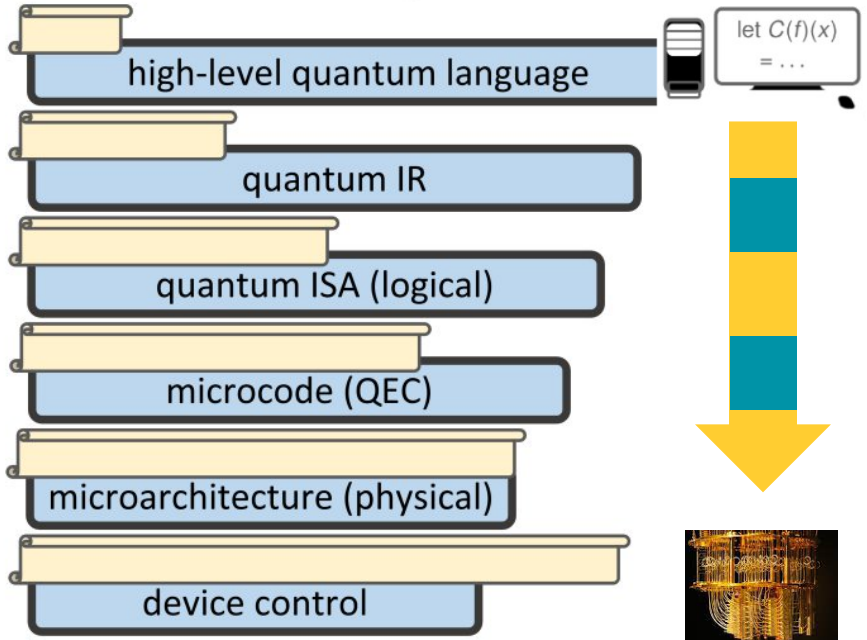




classical computing stack



quantum computing stack




arXiv > quant-ph > arXiv:2211.07629

Search...

Help | Advanced

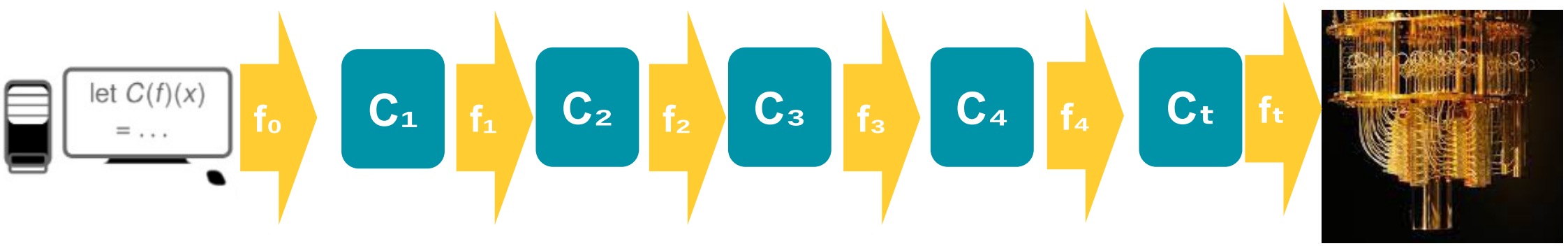
Quantum Physics

[Submitted on 14 Nov 2022]

Assessing requirements to scale to practical quantum advantage

Michael E. Beverland, Prakash Murali, Matthias Troyer, Krysta M. Svore, Torsten Hoefler, Vadym Kliuchnikov, Guang Hao Low, Mathias Soeken, Aarthi Sundaram, Alexander Vaschillo

Guaranteeing the compilation stack



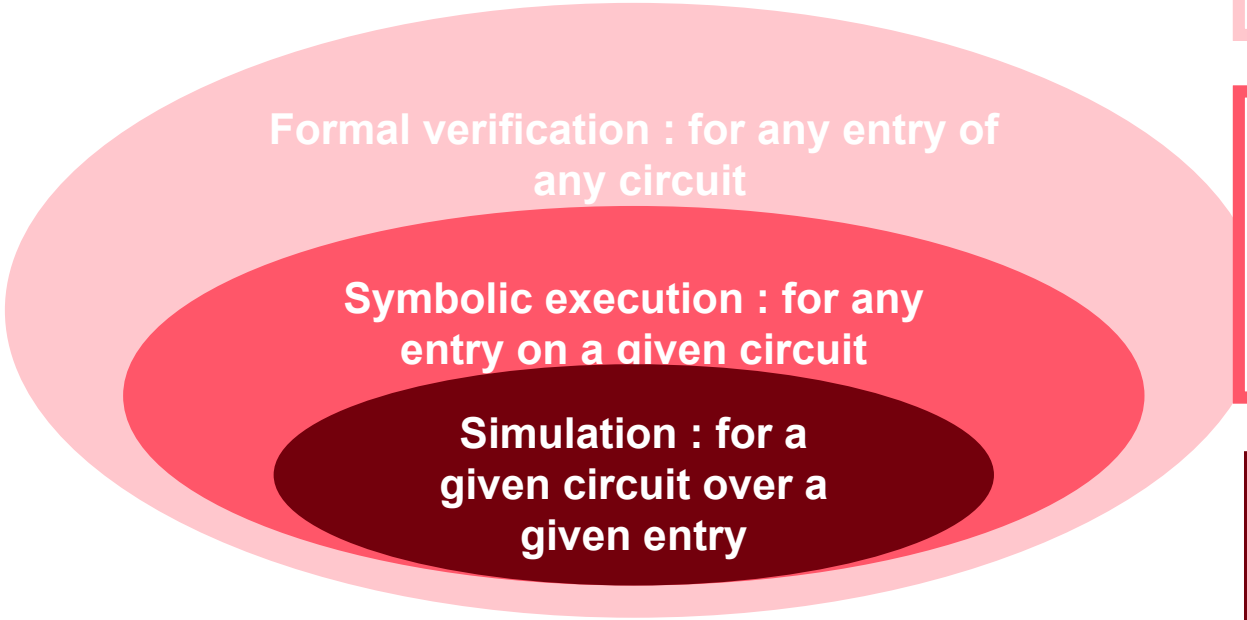
Final circuit:

- Which **physical operations** is it based on ?
- Does it respect the target **topological constraints**?
- **Optimization**?
  - How efficiently?
  - **Metrics**?
- Functional (quasi) **equivalence** wrt C_0
  - Which notion of **distance**?
- Does it include **error correction**?
  - How **robustly**?

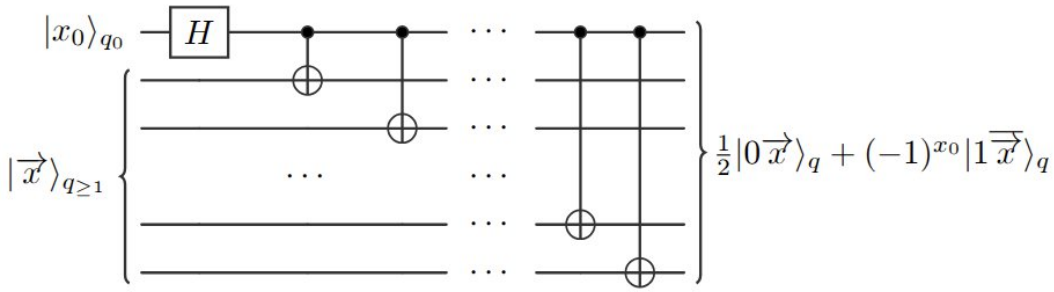
{S}

- $\forall C.S(C_0, f_t \circ \ldots \circ f_4 \circ f_3 \circ f_2 \circ f_1 \circ f_0(C_0))$
  - many f's are hardware dependent
- $S(C_0, C_t)$
  - **compilation choices agnostic**

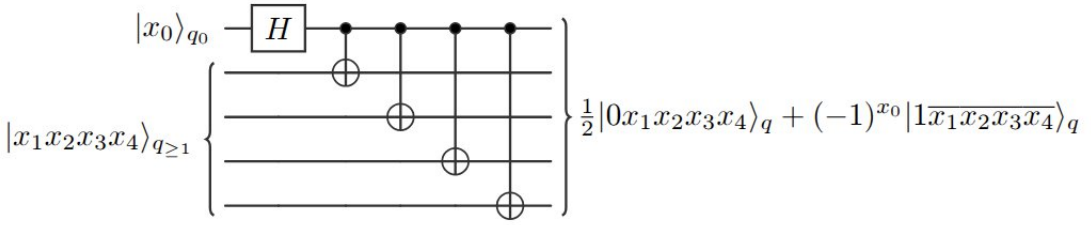
Abstracting parameters/input datas



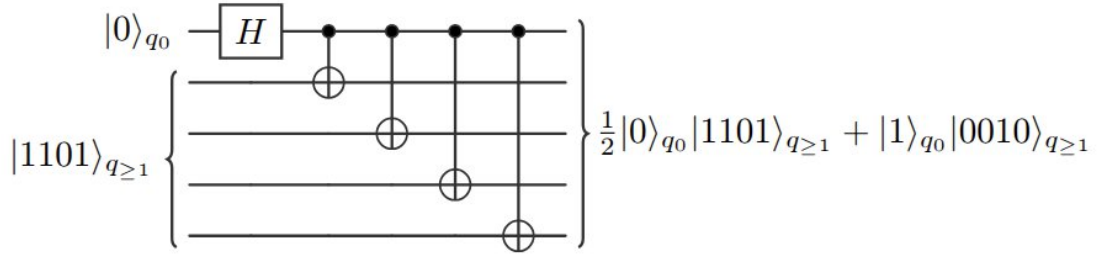
Program level formal proof (forall $x_0x_1x_2x_3x_4$ and forall n, ...)



Symbolic execution (forall $x_0x_1x_2x_3x_4$, ...)



Simulation



Quantum programming and formal verification (Qbricks)

1. **Specifications and programs requirements in quantum computing**
2. **Hybrid symbolic execution**
3. **Unitary formal specification proof**

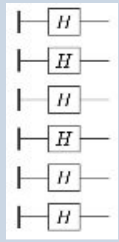
Symbolic execution

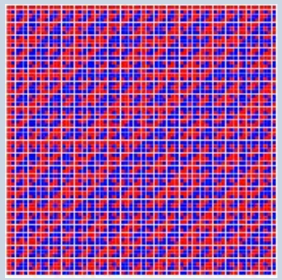
Leading ideas :

- We interprete a program
  - Not as a sequence of quantum physics operations
  - But as instructions operating on a mathematical representation : *semantics*
- Requires :
  - An operational semantics
  - Corresponding data structures
  - Object/meta language
- Program input may remain symbolic in the analysis: derive an i/o function from a program

Programs representation → path-sums

Standard interpretation as matrices :





- Cumbersome
- Requires **higher-order reasoning**

Programs representation → path-sums

Algorithm: Quantum order-finding

Inputs: (1) A black box $U_{x,N}$ which performs the transformation $|j\rangle|k\rangle \rightarrow |j\rangle|x^j k \bmod N\rangle$, for x co-prime to the L-bit number N, (2) $t = 2L + 1 + \left\lceil \log\left(2 + \frac{1}{2\epsilon}\right)\right\rceil$ qubits initialized to $|0\rangle$, and (3) L qubits initialized to the state $|1\rangle$.

Outputs: The least integer $r > 0$ such that $x^r = 1 \pmod N$.

Runtime: $O(L^3)$ operations. Succeeds with probability $O(1)$.

Procedure:

1.	$\|0\rangle\|1\rangle$	initial state
2.	$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} \|j\rangle\|1\rangle$	create superposition
3.	$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} \|j\rangle\|x^j \bmod N\rangle$ $\approx \frac{1}{\sqrt{r2^t}} \sum_{s=0}^{r-1} \sum_{j=0}^{2^t-1} e^{2\pi i s j/r} \|j\rangle\|u_s\rangle$	apply $U_{x,N}$
4.	$\rightarrow \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} \|\widetilde{s/r}\rangle\|u_s\rangle$	apply inverse Fourier transform to first register
5.	$\rightarrow \widetilde{s/r}$	measure first register
6.	$\rightarrow r$	apply continued fractions algorithm

Shor-OF (from N & C, p. 232)

Body :

- An intertwinned sequence of intermediate **state representations**
- A list of **function** applications **declarations**

Programs representation → path-sums

Algorithm: Quantum order-finding

Inputs: (1) A black box $U_{x,N}$ which performs the transformation $|j\rangle|k\rangle \rightarrow |j\rangle|x^j k \bmod N\rangle$, for x co-prime to the L-bit number N, (2) $t = 2L + 1 + \lceil \log(2 + \frac{1}{2\epsilon}) \rceil$ qubits initialized to $|0\rangle$, and (3) L qubits initialized to the state $|1\rangle$.

Outputs: The least integer $r > 0$ such that $x^r = 1 \pmod N$.

Runtime: $O(L^3)$ operations. Succeeds with probability $O(1)$.

Procedure:

1.	$\|0\rangle\|1\rangle$	initial state
2.	$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} \|j\rangle\|1\rangle$	create superposition
3.	$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} \|j\rangle\|x^j \bmod N\rangle \approx \frac{1}{\sqrt{r2^t}} \sum_{s=0}^{r-1} \sum_{j=0}^{2^t-1} e^{2\pi i s j/r} \|j\rangle\|u_s\rangle$	apply $U_{x,N}$
4.	$\rightarrow \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} \|\widetilde{s/r}\rangle\|u_s\rangle$	apply inverse Fourier transform to first register
5.	$\rightarrow \widetilde{s/r}$	measure first register
6.	$\rightarrow r$	apply continued fractions algorithm

Shor-OF (from N & C, p. 232)

$$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|x^j \bmod N\rangle$$

$$\approx \frac{1}{\sqrt{r2^t}} \sum_{s=0}^{r-1} \sum_{j=0}^{2^t-1} e^{2\pi i s j/r} |j\rangle|u_s\rangle$$

Body :

- An intertwinned sequence of intermediate **state representations**
- A list of **function** applications **declarations**

Programs representation → path-sums

$$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle |x^j \bmod N\rangle$$

$$\approx \frac{1}{\sqrt{r2^t}} \sum_{s=0}^{r-1} \sum_{j=0}^{2^t-1} e^{2\pi i s j/r} |j\rangle |u_s\rangle$$

Path-sum semantics[Amy 2019]: **formalizing** vector based circuit representations

$$|x\rangle \rightarrow \frac{1}{\sqrt{2^r}} \sum_{y \in BV_r} e^{i\,\pi\, ph(x,y)}\, |k(x,y)\rangle$$

```
r   : int
ph  : Polynomial of symbolic dyadic fractions
k   : Symbolic binary functions
```

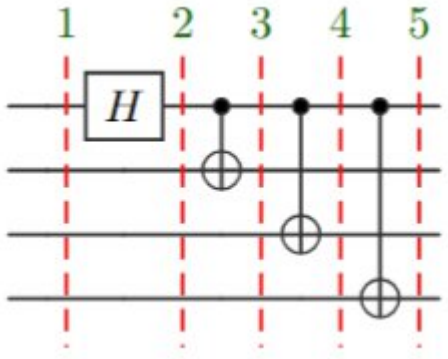
Symbolic execution

– Gate interpretation

$$\langle \mathbf{s}, \mathbf{ph}, \mathbf{o} \rangle \xrightarrow{\mathtt{H}_i} \langle \mathbf{s}+1, \mathbf{ph} + \tfrac{x_i y_f}{2}, \mathbf{o}[x_i \leftarrow y_f] \rangle \xrightarrow{\mathtt{Concr}} \frac{1}{\sqrt{2}} \begin{pmatrix} \mathtt{Concr}(\langle \mathbf{s}, \mathbf{ph}, \mathbf{o} \rangle [\mathrm{x}_i \leftarrow 0]) \\ + (\text{-}1)^{x_i} \\ \mathtt{Concr}(\langle \mathbf{s}, \mathbf{ph}, \mathbf{o} \rangle [\mathrm{x}_i \leftarrow 1]) \end{pmatrix}$$

$$\langle \mathbf{s}, \mathbf{ph}, \mathbf{o} \rangle \xrightarrow{\mathtt{CNOT}_{ij}} \langle \mathbf{s}, \mathbf{ph}, \mathbf{o}[x_j \leftarrow (x_i \oplus x_j)] \rangle \xrightarrow{\mathtt{Concr}} \frac{1}{\sqrt{2^s}} e^{i\pi \mathbf{ph}} |\mathbf{o}[x_j \leftarrow (x_i \oplus x_j)]\rangle$$

– sequential composition



Step	Path-sum	Concretization
1	$\langle 0, 1, x_0x_1x_2x_3 \rangle$	$\lvert x_0x_1x_2x_3 \rangle$
2	$\langle 1, \frac{x_0y}{2}, yx_1x_2x_3 \rangle$	$\frac{1}{\sqrt{2}} (\lvert 0x_1x_2x_3 \rangle_+ \lvert 1x_1x_2x_3 \rangle)$
3	$\langle 1, \frac{x_0y}{2}, y(y \oplus x_1)x_2x_3 \rangle$	$\frac{1}{\sqrt{2}} (\lvert 0x_1x_2x_3 \rangle_+ \lvert 1\overline{x_1}x_2x_3 \rangle)$
4	$\langle 1, \frac{x_0y}{2}, y(y \oplus x_1)(y \oplus x_2)x_3 \rangle$	$\frac{1}{\sqrt{2}} (\lvert 0x_1x_2x_3 \rangle_+ \lvert 1\overline{x_1x_2}x_3 \rangle)$
5	$\langle 1, \frac{x_0y}{2}, y(y \oplus x_1)(y \oplus x_2)(y \oplus x_3) \rangle$	$\frac{1}{\sqrt{2}} (\lvert 0x_1x_2x_3 \rangle_+ \lvert 1\overline{x_1x_2x_3} \rangle)$

– Further constructs (quantum/classical control, parallelism, ...)

Programs representation → path-sums

++ :

- **Modular reasoning** : sequence/parallelism
- **Equational theory** : Path-sum equivalence
  → reduces the exponential factor **r**

$$\text{Elim}\ \frac{y_0 \notin \mathbf{p}, \mathbf{o}}{\langle \mathbf{s}+2, \mathbf{p}, \mathbf{o}\rangle_{\cup\{0\}} \langle \mathbf{s}, \mathbf{p}, \mathbf{o}\rangle}$$

$$\text{HH}\ \frac{y_0 \notin \mathbf{o}, Q, R \qquad y_1 \notin Q}{\langle \mathbf{s}, \tfrac{1}{2}(y_0(y_1+Q)+R), \mathbf{o}\rangle_{\cup\{0\}} \langle \mathbf{s}, R[y_1 \leftarrow \overline{Q}], \mathbf{o}[y_1 \leftarrow \overline{Q}]\rangle}$$

$$\omega\ \frac{y_0 \notin \mathbf{o}, Q, R}{\langle \mathbf{s}+1, \tfrac{1}{4}y_0 + \tfrac{1}{2}y_0 Q + R, \mathbf{o}\rangle_{\cup\{0\}} \langle \mathbf{s}, \tfrac{1}{8} - \tfrac{1}{4}\overline{Q} + R, \mathbf{o}\rangle}$$

Rewriting rule: focus on the HH rule, destructive interference

Consider path-sum

$$\left\langle \mathsf{s}, \frac{1}{2}(y_0(y_1 + Q(\overrightarrow{y})) + R(y_1, \overrightarrow{y})), \mathsf{o}(y_1, \overrightarrow{y}) \right\rangle$$

It is an encoding for (its *concretization*) state

$$\frac{1}{\sqrt{2^{\mathsf{s}}}} \sum_{y_0, y_1, \overrightarrow{y} \in \{0,1\}} e^{2i\pi \frac{1}{2}(y_0(y_1 + Q(\overline{y})) + R(y_1, \overline{y}))} |\mathsf{o}(y_1, \overline{y})\rangle$$

Now, fix the value for every path variable in $y_1, \overline{y}$ (we have two paths left, varying upon y_0):

- If $y_1 ! = Q(\overline{y})$ then $y_0(y_1 + Q(\overline{y})) = y_0$ and these two path differ only by a $e^{\pi i} = -1$ scalar $\rightarrow$ destructive interference
- Otherwise $y_1 = Q(\overline{y})$ then $y_0(y_1 + Q(\overline{y})) = 0$ and both pathes reduce to the same:

$$e^{2i\pi \frac{1}{2} R[y_1 \leftarrow \overline{Q}]} |\mathsf{o}[y_1 \leftarrow \overline{Q}]\rangle$$

$$\text{Elim} \frac{y_0 \notin \mathsf{p}, \mathsf{o}}{\langle \mathsf{s}+2, \mathsf{p}, \mathsf{o}\rangle_{\cup\{0\}} \langle \mathsf{s}, \mathsf{p}, \mathsf{o}\rangle}$$

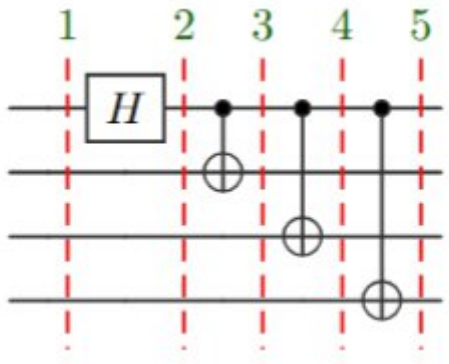
$$\text{HH} \frac{y_0 \notin \mathsf{o}, Q, R \qquad y_1 \notin Q}{\langle \mathsf{s}, \frac{1}{2}(y_0(y_1 + Q) + R), \mathsf{o}\rangle_{\cup\{0\}} \langle \mathsf{s}, R[y_1 \leftarrow \overline{Q}], \mathsf{o}[y_1 \leftarrow \overline{Q}]\rangle}$$

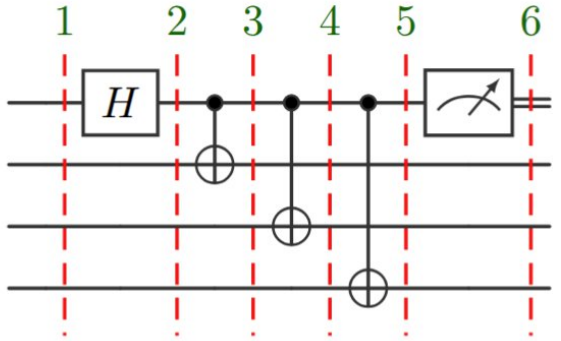
$$\omega \frac{y_0 \notin \mathsf{o}, Q, R}{\langle \mathsf{s}+1, \frac{1}{4}y_0 + \frac{1}{2}y_0 Q + R, \mathsf{o}\rangle_{\cup\{0\}} \langle \mathsf{s}, \frac{1}{8} - \frac{1}{4}\overline{Q} + R, \mathsf{o}\rangle}$$

Symbolic execution : extension to the hybrid case

Measurement is non deterministic:

- **For every possible measurement result i, consider projector $|i><i|$, then**
- **$|x>$ leads to $|i><i||x>$ with probability $|||i><i||x>|^2$**
- **$\rightarrow$ Requires projections, given for free in path-sums**

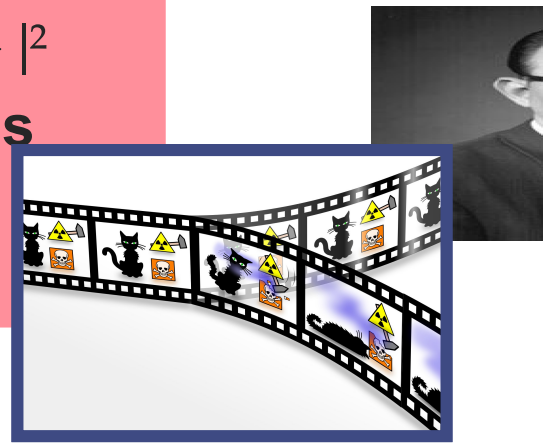




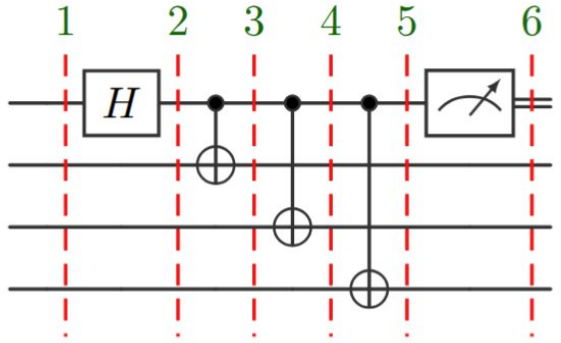
Symbolic execution : extending to the hybrid case

Measurement is non deterministic:

- **For every possible measurement result i, consider projector $|i><i|$, then**
- **$|x>$ leads to $|i><i||x>$ with probability $|||i><i||x>|^2$**
- **$\rightarrow$ Requires projections, given for free in path-sums**
  - **Identify a data cell as a *classical* one**
  - **Then the concretization differentiates**



Step	Path-sum	Concretization
1	$\langle 0, 1, \lvert x_0x_1x_2x_3\rangle_q\rangle$	$\lvert x_0x_1x_2x_3\rangle_q$
2	$\langle 1, \frac{x_0y}{2}, \lvert yx_1x_2x_3\rangle_q\rangle$	$\frac{1}{\sqrt{2}}(\lvert 0x_1x_2x_3\rangle_q + \lvert 1x_1x_2x_3\rangle_q)$
3	$\langle 1, \frac{x_0y}{2}, \lvert y(y\oplus x_1)x_2x_3\rangle_q\rangle$	$\frac{1}{\sqrt{2}}(\lvert 0x_1x_2x_3\rangle_q + \lvert 1\overline{x_1}x_2x_3\rangle_q)$
4	$\langle 1, \frac{x_0y}{2}, \lvert y(y\oplus x_1)(y\oplus x_2)x_3\rangle_q\rangle$	$\frac{1}{\sqrt{2}}(\lvert 0x_1x_2x_3\rangle_q + \lvert 1\overline{x_1x_2}x_3\rangle_q)$
5	$\langle 1, \frac{x_0y}{2}, \lvert y(y\oplus x_1)(y\oplus x_2)(y\oplus x_3)\rangle_q\rangle$	$\frac{1}{\sqrt{2}}(\lvert 0x_1x_2x_3\rangle_q + \lvert 1\overline{x_1x_2x_3}\rangle_q)$
6	$\langle 1, \frac{x_0y}{2}, [y]_{q_0}\lvert (y\oplus x_1)(y\oplus x_2)(y\oplus x_3)\rangle_{q\geq 1}\rangle$	$\left\{\begin{array}{ll}\frac{1}{2} & ;[0]_{q_0}, \lvert x_1x_2x_3\rangle_{q\geq 1}\\ \frac{1}{2} & ;[1]_{q_0}, \lvert\overline{x_1x_2x_3}\rangle_{q\geq 1}\end{array}\right\}$



Symbolic execution : extending to the hybrid case

Consequences on the calculus:

- **Specific constraint for unitary rules**
  should not compute interference for paths not communicating anymore
- **Unbalanced (norm varying) paths sums**
- **Further reduction and transformation**
  - **Eg. : state separation states and register dicard (see ex. Teleportation)**

Step	Path-sum	Concretization
1	$\langle 0, 1, \lvert x_0x_1x_2x_3\rangle_q\rangle$	$\lvert x_0x_1x_2x_3\rangle_q$
2	$\langle 1, \frac{x_0y}{2}, \lvert yx_1x_2x_3\rangle_q\rangle$	$\frac{1}{\sqrt{2}}(\lvert 0x_1x_2x_3\rangle_q + \lvert 1x_1x_2x_3\rangle_q)$
3	$\langle 1, \frac{x_0y}{2}, \lvert y(y\oplus x_1)x_2x_3\rangle_q\rangle$	$\frac{1}{\sqrt{2}}(\lvert 0x_1x_2x_3\rangle_q + \lvert 1\overline{x_1}x_2x_3\rangle_q)$
4	$\langle 1, \frac{x_0y}{2}, \lvert y(y\oplus x_1)(y\oplus x_2)x_3\rangle_q\rangle$	$\frac{1}{\sqrt{2}}(\lvert 0x_1x_2x_3\rangle_q + \lvert 1\overline{x_1x_2}x_3\rangle_q)$
5	$\langle 1, \frac{x_0y}{2}, \lvert y(y\oplus x_1)(y\oplus x_2)(y\oplus x_3)\rangle_q\rangle$	$\frac{1}{\sqrt{2}}(\lvert 0x_1x_2x_3\rangle_q + \lvert 1\overline{x_1x_2x_3}\rangle_q)$
6	$\langle 1, \frac{x_0y}{2}, [y]_{q_0} \lvert (y\oplus x_1)(y\oplus x_2)(y\oplus x_3)\rangle_{q\geq 1}\rangle$	$\left\{ \begin{array}{ll} \frac{1}{2} & ; [0]_{q_0}, \lvert x_1x_2x_3\rangle_{q\geq 1} \\ \frac{1}{2} & ; [1]_{q_0}, \lvert \overline{x_1x_2x_3}\rangle_{q\geq 1} \end{array} \right\}$

Hybrid SE : the QBricks tool

- Embedding in Ocaml
  - a «hybrid program type »
- + a hybrid path-sum data type
  - execution representation + rewriting
- prog + params are interpreted as hybrid path sums
- various commands for
  - program to path-sum evaluation
  - automated/interactive rewriting
  - specification verification (equality + refinment)

  ~ unitary equivalence verif
- post treatment availabilities from the hosting language (Ocaml)

```
type prog =
    | PVar of string
    | Skip
    | InitQReg of qreg
    | Gate of gate * qreg
    | Meas of qreg * creg
    | Seq of prog * prog
    | SetCReg of creg * pr_int
    | For of string * pr_int * pr_int * prog
    | IfElse of pr_bool * prog * prog
```

```
type gate = H | X | Z | Rz of Prog.Common_types.pr_int
```

```
<
  0
  ,
  sqrt(1 / 2)
  .
  |y[0]>_q0[0] |y[0]>_q1[0]
  [
    -
  ]
>{y0}
```

```
Rewrite: hh -find
No possible hh found
```

Hands on session

- Goals :
  - write a couple standard circuits, from gate to gate application to Quantum Fourier Transform
  - generate and read hybrid path sums
  - illustrate rewritings (interactive/automated)
  - experiment hybrid circuit verification

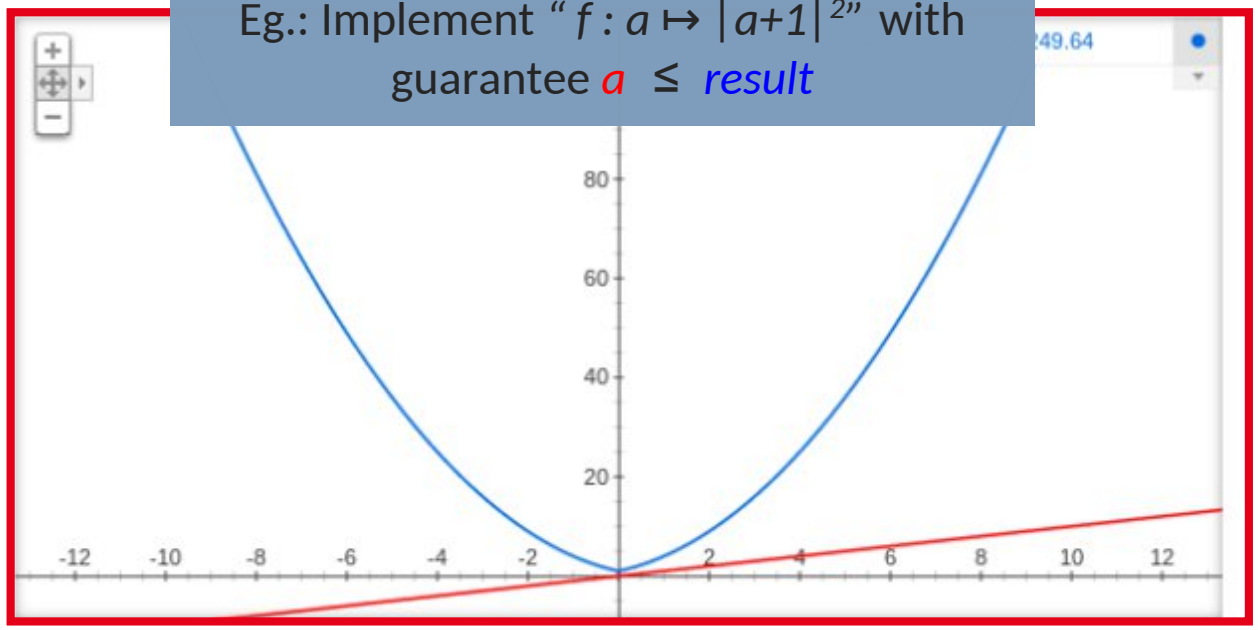
- Content :
  - Bell state and GHZ generalization
  - quantum teleportation protocol
  - a glimpse of quantum error correction
  - the Quantum Fourier Transform

Quantum programming and formal verification (Qbricks)

1. **Specifications and programs requirements in quantum computing**
2. **Hybrid symbolic execution**
3. **Unitary formal specification proof**

A parte: introducing deductive verification

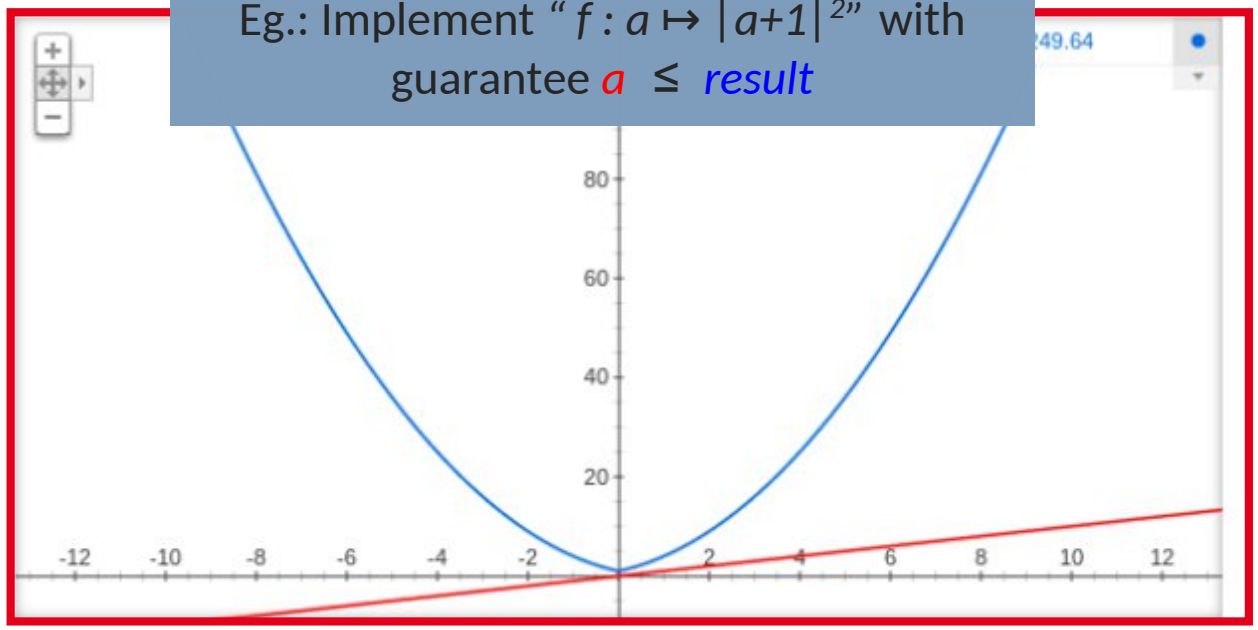
Eg.: Implement “ $f : a \mapsto |a+1|^2$ ” with guarantee $a \leq result$



A parte: introducing deductive verification

Eg.: Implement “ $f : a \mapsto |a+1|^2$” with guarantee $a \leq result$

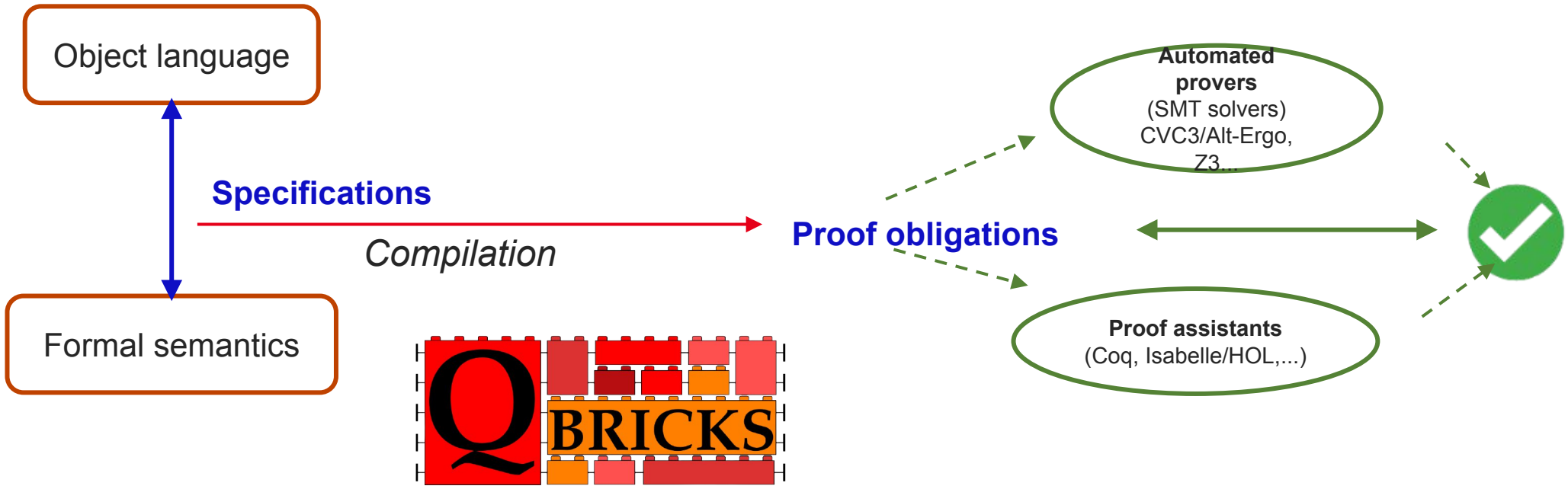
```
let abs_succ (a):
    post{result ≥1}
    = |a+1|

let mult (a b)
    post{a ≥ 1 → result ≥ b }
    ...
    = a × b

def sq_abs_succ (a):
    ensures{result ≥ a}
    = mult (abs_succ(a),abs_succ(a))
```



Deductive verification scheme

- Deductive verification : annotate (object language) code with (formal language) path-sum components specifications
  - preconditions
  - postconditions
  - loop invariants ...

- Called functions bring their specifications as guaranteed theorems for proving the specs of calling functions



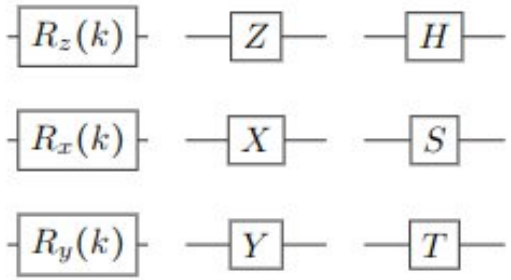
Qbricks : object language

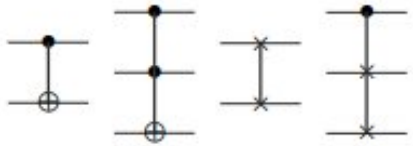
- **Functional** programming + Why3 embedding
- Circuit as objects → **no cloning by construction**
- **Deductive verification** → use of **contracts (well formedness + functional correctness + complexity)**
- Unitary programs

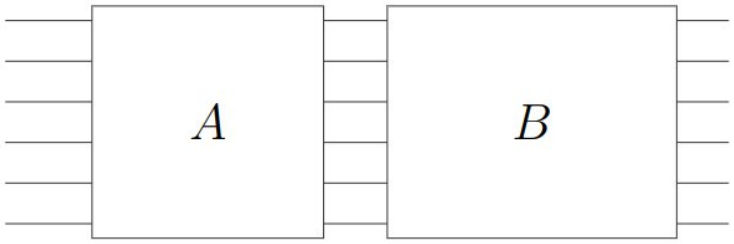
- A set of **Elementary gates**
- Some quantum **circuit combinators**

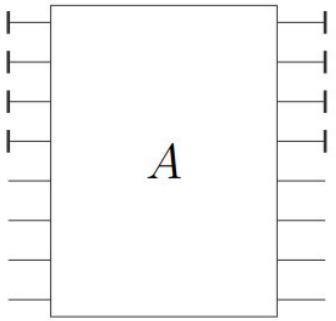
- Derived **high-level combinators**: inversion, qbit permutations, etc

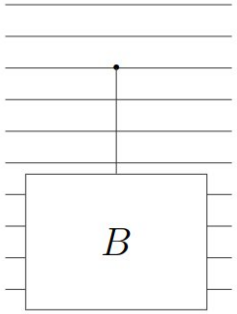
- **Inheritance** from Whyml:
  - Let construct
  - Loops
  - References...

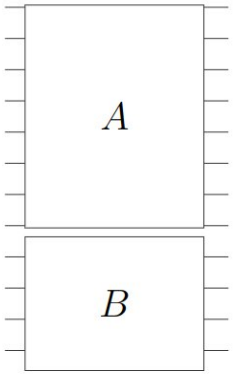






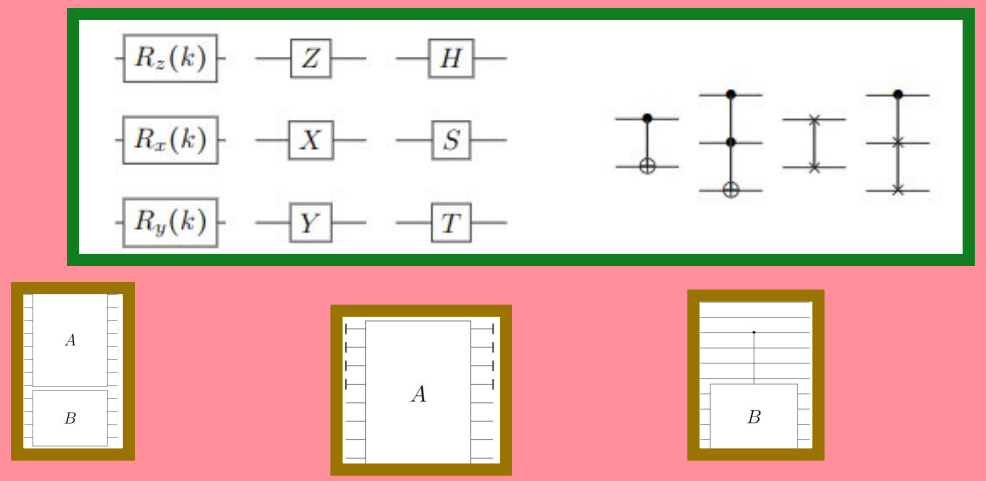






Specifying Qbricks code

Type «pre_circuit » : algebraic structure

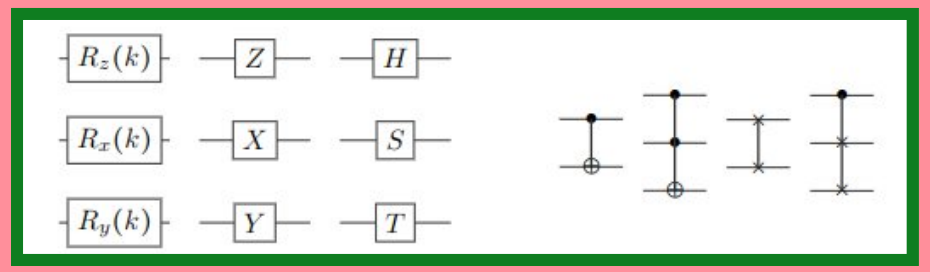


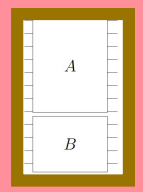

```
type pre_circuit =
  Skip |  Phase int | Rx int | Ry int | Rz int | Rzp int | Hadamard  |  S | T | X | Y | Z
  | Bricks_Cnot | Bricks_Toffoli | Bricks_Fredkin| Bricks_Swap
  | Swap int int int | Cnot int int int | Toffoli int int int | Fredkin int int int
  | Place wired_circuit int int
  | Cont wired_circuit int int int
  | Sequence wired_circuit wired_circuit
  | Parallel wired_circuit wired_circuit
  | Ancillas wired_circuit int
```

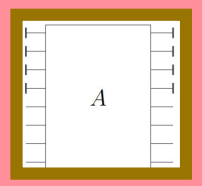
Specifying Qbricks code

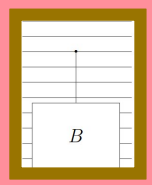
Type «pre_circuit » : algebraic structure

Some of them are « correct »
→ eg : instances of control respect unitarity







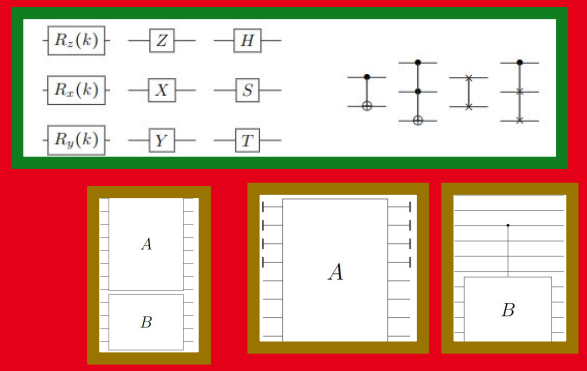



```
let rec predicate build_correct (c:wired_circuit)
  =  match c with
     Cnot c t n ->  0<= c <n&& 0<= t <n&& c<> t
   | Swap t1 t2 n -> 0<= t1<n && 0<= t2 <n&& t1<> t2
   | Toffoli c1 c2 t n ->  0<= c1<n && 0<= c2 <n&& 0<= t <n&& c1 <> t && c2 <> t && c1 <> c2
   | Fredkin c1 c2 t n ->  0<= c1<n && 0<= c2<n && 0<= t <n&& c1 <> t && c2 <> t && c1 <> c2
   | Place c t n ->  build_correct c && 0<=t<n && t+width_pre c <=n
   | Cont c co t n ->  build_correct c && 0<= co <n && 0<= t <= n - width_pre c && (  co<t || t + width_pre c <= co)
   | Sequence d e -> build_correct d  && build_correct e && width_pre d =  width_pre e
   | Parallel d e ->  build_correct d  && build_correct e
   | Ancillas d i ->  build_correct d && 1<=i && i+1 <= width_pre d
   | _ -> true
end
```

Specifying Qbricks code

Type «pre_circuit » :
algebraic structure

Some of them are « correct »
→ eg : instances of control respect unitarity




```
let rec predicate build_correct (c:wired_circuit)
  =  match c with
     Cnot c t n ->  0<= c <n&& 0<= t <n&& c<> t
   | Swap t1 t2 n -> 0<= t1<n && 0<= t2 <n&& t1<> t2
   | Toffoli c1 c2 t n ->  0<= c1<n && 0<= c2 <n&& 0<= t <n&& c1 <> t && c2 <> t && c1 <> c2
   | Fredkin c1 c2 t n ->  0<= c1<n && 0<= c2<n && 0<= t <n&& c1 <> t && c2 <> t && c1 <> c2
   | Place c t n ->  build_correct c && 0<=t<n && t+width_pre c <=n
   | Cont c co t n ->  build_correct c && 0<= co <n && 0<= t <= n - width_pre c && (  co<t || t + width_pre c <= co)
   | Sequence d e -> build_correct d  && build_correct e && width_pre d =  width_pre e
   | Parallel d e ->  build_correct d  && build_correct e
   | Ancillas d i ->  build_correct d && 1<=i && i+1 <= width_pre d
   | _ -> true
end
```

Specifying Qbricks code

Some
→ e
respe

```
let rec function basis_ket (c:circuit) (x y : int-> int) (i:int): int
      variant{pre c}
      ensures{result = h_basis_ket (circ_to_pps_pre(pre c)) x y i}
      ensures{ (forall i:int. 0<= i < width c -> 0<= x i < 2) ->
               (forall i:int. 0<= i < range c -> 0<= y i < 2) ->
                      0<= i < width c -> 0<=result <2}
 = match (pre c) with
   Skip -> x i
  |Phase _ -> x i
  | Rx _ -> y 1
  | Ry _ -> y 1
  | Rz _ -> x i
  | Rzp _ -> x i
  | Hadamard -> y i
  | S -> x i
  | T -> x i
  | X -> 1- x i
  | Y -> 1 - x i
  | Z -> x i
  | Bricks_Cnot -> if i = 1  then x 0 *  (1 - x 1) + x 1 *  (1 - x 0) else x i
  | Bricks_Toffoli  -> if i = 2 then x 0 * x 1 *  (1 - x 2) + x 2 *  (1 - (x 0 * x 1)) else x i
  | Bricks_Fredkin   ->
     if i = 1 then x 0 * x 2 + (1- x 0) * x 1
     else if i = 2 then x 0 * x 1 + (1- x 0) * x 2
     else x i
  | Bricks_Swap   -> if i = 0 then x 1 else if i = 1 then x 0 else  x i
  | Cnot c t _ ->  if i = t  then x c *  (1 - x t) + x t *  (1 - x c) else x i
  | Toffoli c1 c2 t _ -> if i = t then x c1 * x c2 *  (1 - x t) + x t *  (1 - (x c1 * x c2)) else x i
  | Fredkin c t1 t2 _ ->
     if i = t1 then x c * x t2 + (1- x c) * x t1
```

_function ;

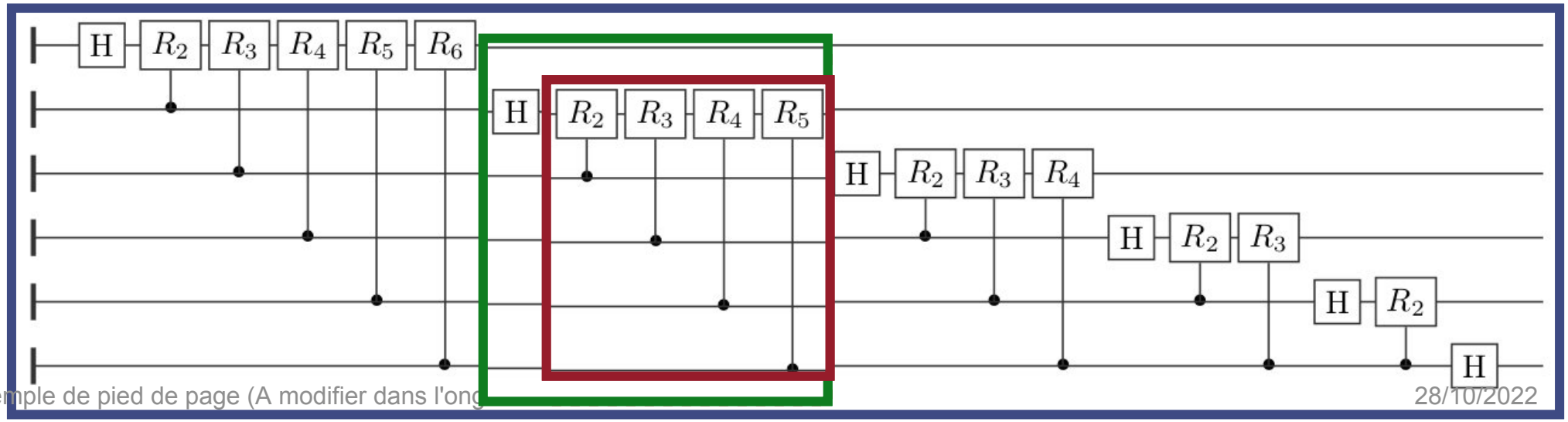
Specifying Qbricks code : constructors

```
let constant hadamard : circuit
  = {pre = Hadamard}
      ensures{pre result = Hadamard}
      ensures{ancillas  result = 0}
      ensures{size result = 1}
      ensures{range result = 1}
      ensures{width result = 1}
      ensures{forall x y: int->int. forall i:int. basis_ket result x y i = y i}
      ensures{forall x y: int->int. ang_ind result x y  = (x 0 *y 0)/./1 }
```

```
let function (--) ( d e:circuit)
    requires{width d = width e}
    ensures{pre result = Sequence (pre d) (pre e)}
    ensures{ancillas  result = max (ancillas d)  (ancillas e) }
    ensures{size result = size d + size e}
    ensures{width result = width d}
    ensures{range result = range d + range e}
    ensures{forall x y: int->int. forall i:int. basis_ket result x y i =
           basis_ket e (basis_ket d x y) (fun k -> y (k+ range d)) i}
    ensures{forall x y: int->int. ang_ind result x y  =
           (ang_ind d x y) +.+ (ang_ind e (basis_ket d x y)  (fun k -> y (k+ range d)))}
  = {pre  = Sequence (pre d) (pre e)}
```

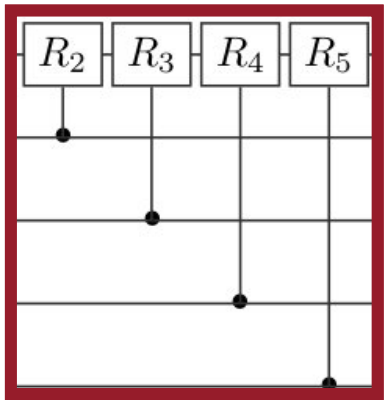
Qbricks at work : the quantum Fourier transform

$$|x\rangle \longrightarrow \frac{1}{\sqrt{2}^n} \sum_{y \in [0,2^n[} e^{\frac{2i\pi x(-y)}{2^n}} |y\rangle$$



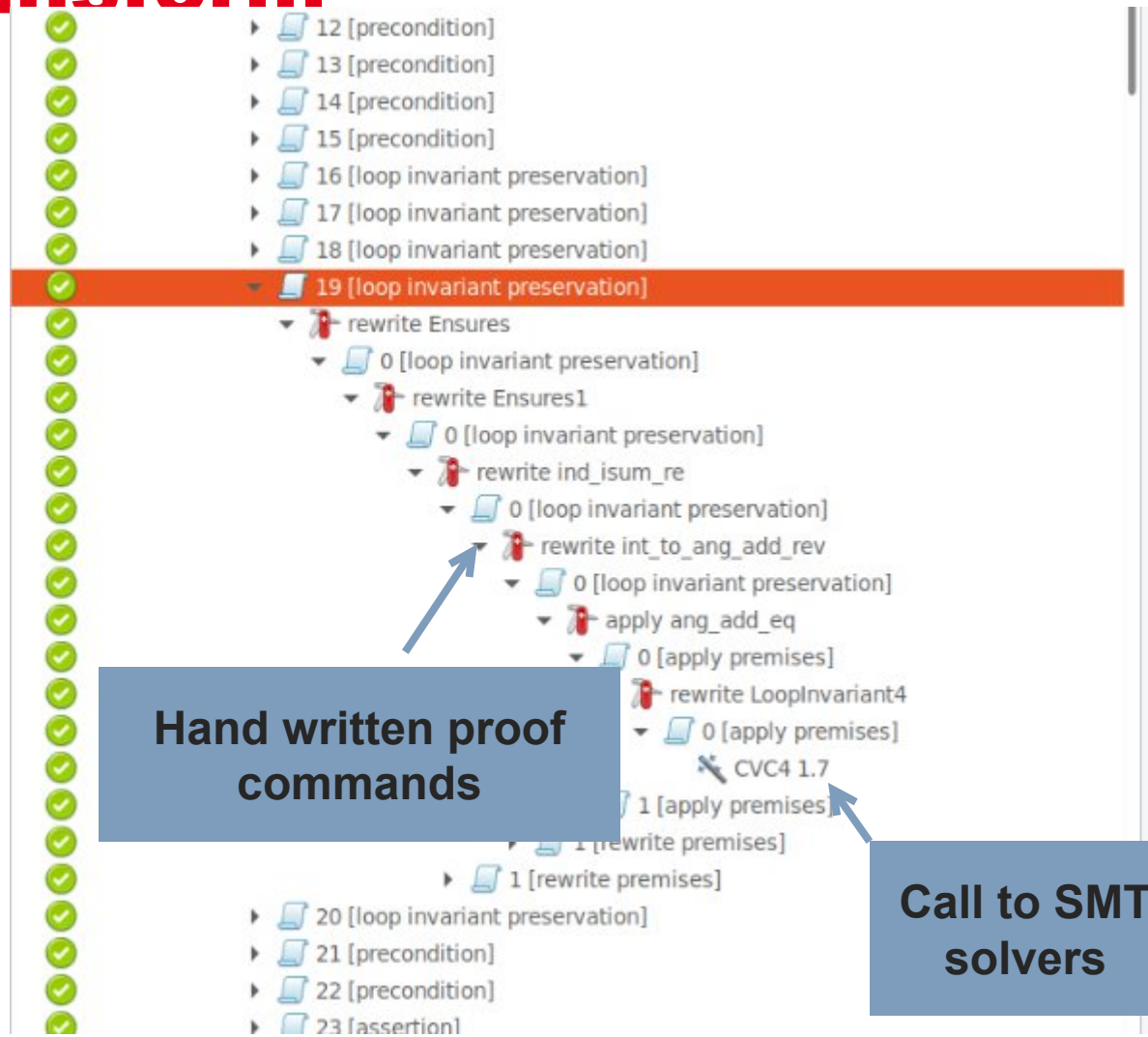
Qbricks at work : the quantum Fourier transform

```
let cl = ref (m_skip n)
in for i = q+1 to n-1 do
     invariant{width !cl = n}
     invariant{range !cl = 0}
     invariant{forall x y i. 0<= i < n ->
                basis_ket !cl x y i = x i}
     invariant{forall x y. ang_ind !cl x y
                =(sum (fun l -> x l * x q * power 2 (n- l -1+ q)) (q+1) i) /./n}
     invariant{size !cl <= cont_size *  (i-q-1) }
     cl := !cl -- (crz i (q) (i - q+1) n );
   done;
```



$$\mathbf{PS}(CRZs_{n,q}) : |x\rangle \longrightarrow \frac{1}{\sqrt{2^0}} \sum_{y \in \mathrm{BV}0} e^{i\pi \frac{\sum_{l=q+1}^{n} (x_l * x_q * 2^{n-l-1+q})}{2^n}} |x\rangle$$

Qbricks at work : the quantum Fourier transform



Hand written proof commands

Call to SMT solvers

```
165
166 Ensures1 :
167   forall x1:int -> int, y1:int -> int.
168    ang_ind (cl1 -- crz i q ((i -'' q) +' 1) n) x1 y1 =
169    (ang_ind cl1 x1 y1
170     +.+ ang_ind (crz i q ((i -'' q) +' 1) n)
171         ((((fun (y0:circuit) (y11:int -> int) (y2:int -> int) (y3:int) ->
172              basis_ket y0 y11 y2 y3)
173            @ cl1)
174           @ x1)
175          @ y1)
176         (fun (k:int) -> y1 @ (k +' range cl1)))
177
178 constant cl : circuit
```

Called functions post ∈ logical context

```
182 LoopInvariant2 : width cl = n
183
184 LoopInvariant1 : range cl = 0
185
186 LoopInvariant :
187   forall x1:int -> int, y1:int -> int, i1:int.
188    0 <=' i1 /\ i1 <' n -> basis_ket cl x1 y1 i1 = (x1 @ i1)
189
190 constant x : int -> int
191
192 constant y : int -> int
193
194 ------------------------------ Goal ------------------------------
195
196 goal qft'vc :
197   ang_ind cl x y =
198   (ind_isum
199    (fun (l:int) -> ((x @ l) *' (x @ q)) *' power 2 (((n -'' l) -'' 1) +' q))
200    (q +' 1) (i +' 1) /./ n)
201
202
```

Post-conditions ⇒ proof obligations

Hands on session

- Goals :
  - experience of circuit building + path-sum specification writing in Qbricks
  - introduce to parametricity and recursive proofs
  - get a sense of proof automation optimization
  - experiment imperative style verified programming, using ref (mutable) types

- Content :
  - Bell state and GHZ generalization : For loop and recursive definition
  - Quantum Fourier Transform

- Starting :
  - download Qbricks at https://github.com/Qbricks/qbricks.github.io
  - create, install docker :
    - make build
    - make container
    - make start
  - cd tutorial_pldi
  - open file tutorial.mlw (tutorial.html) and read introductory comments

Any question ?

Christophe Chareton
Christophe.chareton@cea.fr