Séminaire Gallium
2 décembre 2019

Toward certified quantum programming

Christophe Chareton

Sébastien Bardin, François Bobot, Valentin Perrelle (CEA) and Benoît Valiron (LRI)

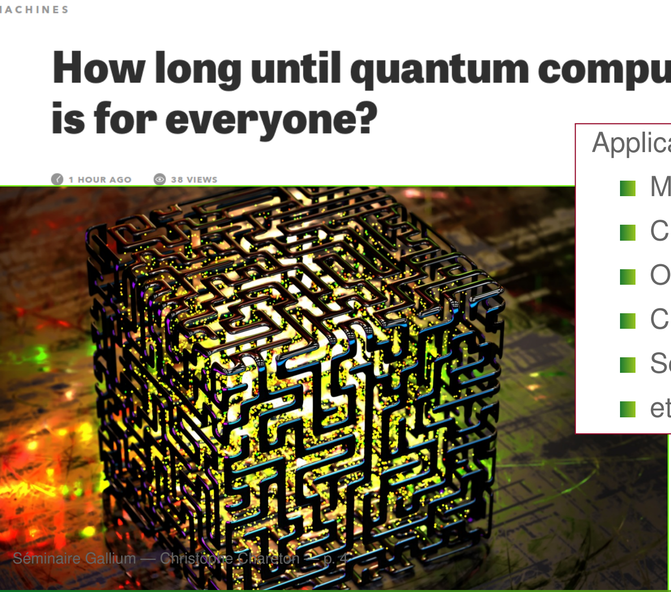
Take away

- Quantum computers (are going to / will . . .) arrive

  → How to write <u>correct</u> programs?
- Need specification and verification mechanisms
  - scale invariant
  - close to quantum algorithm descriptions
  - well distinguished from code itself
  - largely automated
- We are developing *Qbricks* as a first step towards this goal
  - core building circuit language
  - dual semantics
  - high level specification framework
- Certified implementation of the phase estimation algorithm (quantum part of Shor)

Outline

Dreams of quantum computing



Applications:

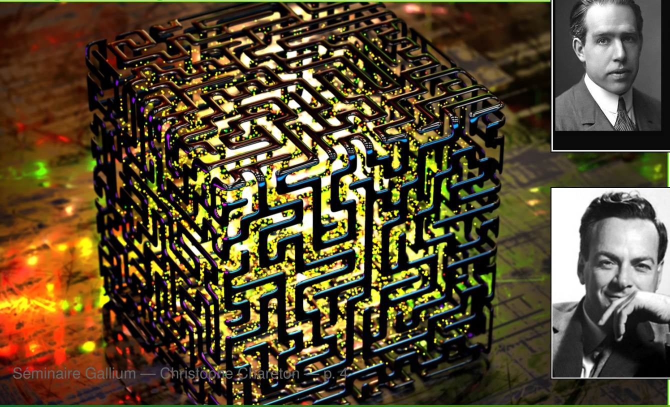
- Machine learning
- Chemistry
- Optimisation
- Cryptography
- Scheduling
- etc

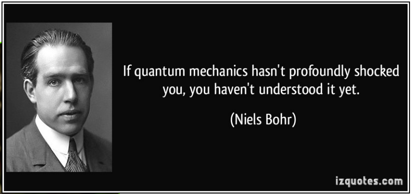
Dreams of quantum computing

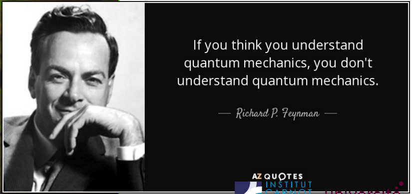
ACHINES

How long until quantum computing is for everyone?

1 HOUR AGO 38 VIEWS







They are coming

IBM will soon launch a 53-qubit quantum computer

Frederic Lardinois @fredericl / 12:00 pm +00 • September 18, 2019 Co

Google claims to have reached quantum supremacy

Researchers say their quantum computer has calculated an impossible problem for ordinary machines

Context

They are coming

IBM will soon launch a 53-qubit quantum computer

Frederic Lardinois @fredericl / 12:00 pm +00 • September 18, 2019

Noisy Intermediate Scale Quantum (NISQ - J. Preskill 2018)

- Algorithm have to deal with noise
- Limited ressources :
  - 50 - 1000 qubits
  - limited circuit depth

Google claims to have reached quantum supremacy

Researchers say their quantum computer has calculated an impossible problem for ordinary machines

They are coming

IBM will soon launch a 53-qubit quantum co

Frederi

Go

sup

Researchers say their quantum computer has calculated an impossible problem for ordinary machines

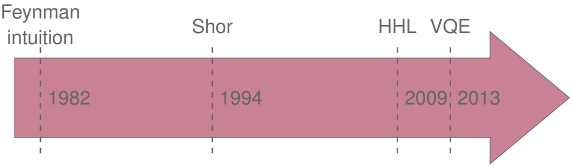
Theoretical results:

- Computing gains Vs best-known classical algorithms:
  - Quantum walks: from linear to logarithmic
  - Grover-based searches: from e^n to $e^{\frac{n}{2}}$
  - Shor-like algorithms: from exponential to linear
  - Quantum simulation: from exponential to linear
- "Absolute" theoretical quantum supremacy?

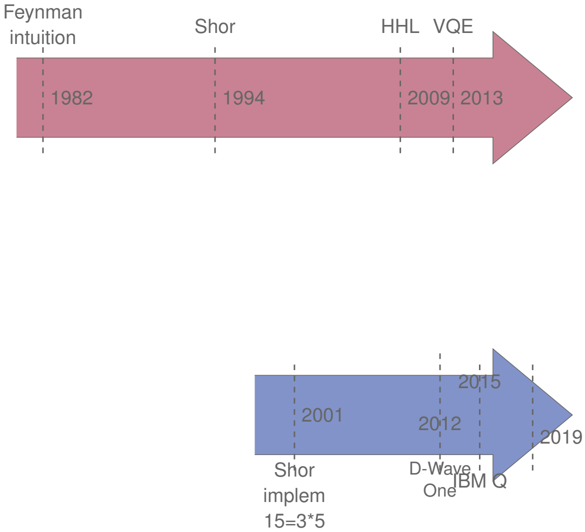
$$\mathbf{BQP} \overset{?}{\subseteq} \mathbf{NP} \qquad \mathbf{BPP} \subseteq \mathbf{BQP}$$

$$\mathbf{BQP} \overset{?}{\supseteq} \mathbf{NP} \qquad \mathbf{BPP} \overset{?}{=} \mathbf{BQP}$$

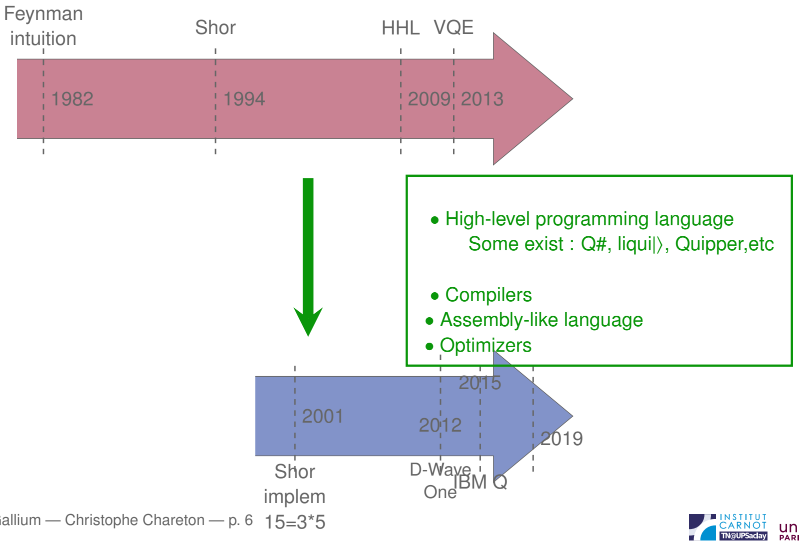
Quantum computing milestone history



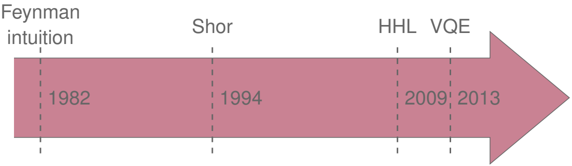
Quantum computing milestone history

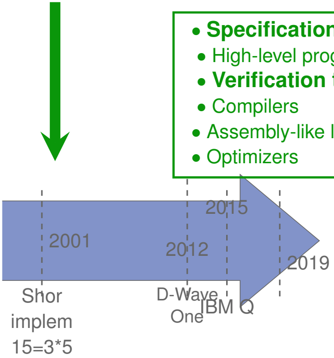


Quantum computing milestone history



Quantum computing milestone history





- **Specification**
  - High-level programming language
  - **Verification tools**
  - Compilers
- Assembly-like language
- Optimizers

Quantum information

- Classical world:

🐱 **XOR** ~~🐱~~

- Quantum world[1]

$$\alpha_0 \text{🐱} \oplus \alpha_1 \cancel{\text{🐱}}$$

with $\alpha_0, \alpha_1 \in \mathbb{C}, |\alpha_0|^2 + |\alpha_1|^2 = 1$

[1] In a 2^n dimension vector space, $|k\rangle_n$ designates the k^{th} canonical basis vector

Quantum information

- Classical world:

  🐱$_0$ 🐱$_1$ ~~🐱~~$_2$ 🐱$_3$ $\cdots$ ~~🐱~~$_{n-1}$

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

- Quantum world[1]

$$|\alpha\rangle_n = \bigoplus \begin{array}{ll} \alpha_0 & \text{🐱}_0\ \text{🐱}_1\ \text{🐱}_2\ \text{🐱}_3 \cdots \text{🐱}_{n-1} \\ \alpha_1 & \text{🐱}_0\ \text{🐱}_1\ \text{🐱}_2\ \text{🐱}_3 \cdots \cancel{\text{🐱}}_{n-1} \\ \cdots & \qquad\cdots \\ \alpha_{2^n-1} & \cancel{\text{🐱}}_0\ \cancel{\text{🐱}}_1\ \cancel{\text{🐱}}_2\ \cancel{\text{🐱}}_3 \cdots \cancel{\text{🐱}}_{n-1} \end{array} = \bigoplus \begin{array}{l} \alpha_0|0\rangle_n \\ \alpha_1|1\rangle_n \\ \cdots \\ \alpha_{2^n-1}|2^{n-1}\rangle_n \end{array} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \cdots \\ \alpha_{2^n-1} \end{pmatrix}$$

[1] In a 2^n dimension vector space, $|k\rangle_n$ designates the k^{th} canonical basis vector

Quantum information

- Classical world:

  🐱$_0$ 🐱$_1$ ~~🐱~~$_2$ 🐱$_3$ $\cdots$ ~~🐱~~$_{n-1}$

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

- Quantum world[1]

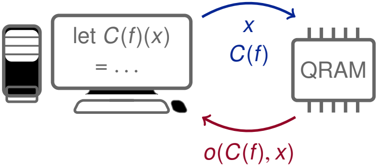
$$|\alpha\rangle_n = \bigoplus \begin{array}{ll} \alpha_0 & 🐱_0\ 🐱_1\ 🐱_2\ 🐱_3 \cdots 🐱_{n-1} \\ \alpha_1 & 🐱_0\ 🐱_1\ 🐱_2\ 🐱_3 \cdots \cancel{🐱}_{n-1} \\ \cdots & \qquad \cdots \\ \alpha_{2^n-1} & \cancel{🐱}_0\ \cancel{🐱}_1\ \cancel{🐱}_2\ \cancel{🐱}_3 \cdots \cancel{🐱}_{n-1} \end{array}$$

+ Some strange rules:
  - no cloning
  - destructive measure
  - operations restricted to unitary

[1] In a 2^n dimension vector space, $|k\rangle_n$ designates the k^{th} canonical basis vector

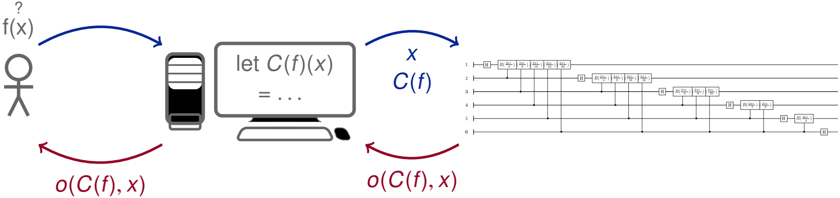
The QRAM model

- A quantum co-processor (QRAM), controlled by a classical computer
  - Classical control flow
  - Quantum computing request, sent to the QRAM
- $\rightarrow$ Structured sequences of instructions: quantum circuits



The QRAM model

- A quantum co-processor (QRAM), controlled by a classical computer
  - Classical control flow
  - Quantum computing request, sent to the QRAM
- → Structured sequences of instructions: quantum circuits



Does the circuit fit the computation need?

Outline

- Context
- The case for verification of quantum algorithms
- *Qbricks*
  - Circuit language
  - Dual semantics
  - Derive proof obligations
  - Toward further automation
- Case study: phase estimation algorithm
- Conclusion

How do we check them?

implements

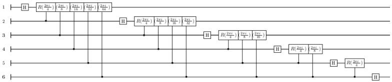
$\|0\rangle\|u\rangle$	initial state
$\to \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}\|j\rangle\|u\rangle$	create superposition
$\to \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}\|j\rangle U^j\|u\rangle$	apply black box
$= \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}e^{2\pi i j\varphi_u}\|j\rangle\|u\rangle$	result of black box
$\to \|\widetilde{\varphi_u}\rangle\|u\rangle$	apply inverse Fourier transform
$\to \widetilde{\varphi_u}$	measure first register

Quantum phase estimation (from Nielsen & Chuang)

```
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')
```

Quipper QFT circuit building function

builds



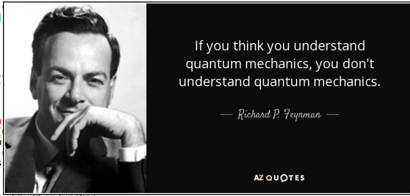
How do we check them?

implements

$\vert 0\rangle \vert u\rangle$	initial state
$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} \vert j\rangle \vert u\rangle$	create superposition
$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} \vert j\rangle U^j \vert u\rangle$	apply black box
$= \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} e^{2\pi i j \varphi_u} \vert j\rangle \vert u\rangle$	result of black box
$\rightarrow \vert \widetilde{\varphi_u}\rangle \vert u\rangle$	apply inverse Fourier transform
$\rightarrow \widetilde{\varphi_u}$	measure first register

Quantum phase estimation (from Nielsen & Chuang)

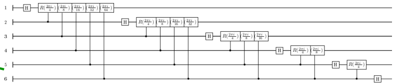
```
qft_internal :: [Qubit] ->
qft_internal [] = return []
qft_internal [x] = do
  hadamard x
  return [x]
qft_internal (x:xs) = do
  xs' <- qft_internal xs
  xs'' <- rotations x xs' (
  x' <- hadamard x
  return (x':xs'')
  where
    -- Auxiliary function u
    rotations :: Qubit -> [
    rotations _ [] _ = retu
    rotations c (q:qs) n =
      qs' <- rotations c qs
      q' <- rGate ((n + 1)
      return (q':qs')
```



Quipper QFT circuit building function

builds

runs
?



- Quantum programming is tricky and non-intuitive
- No means to control an execution
- **Tests are expensive and often statistical**

How do we check them?

implements

$\|0\rangle\|u\rangle$	initial state
$\rightarrow \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}\|j\rangle\|u\rangle$	create superposition
$\rightarrow \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}\|j\rangle U^j\|u\rangle$	apply black box
$= \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1} e^{2\pi i j \varphi_u}\|j\rangle\|u\rangle$	result of black box
$\rightarrow \|\widetilde{\varphi_u}\rangle\|u\rangle$	apply inverse Fourier transform
$\rightarrow \widetilde{\varphi_u}$	measure first register

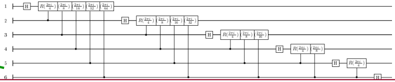
Quantum phase estimation (from Nielsen & Chuang)

```
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')
```

Quipper QFT circuit building function

builds

runs
?



- Testing is difficult . . .
- What about full verification? allows to handle
  - Infinite state space
  - absolute guarantee

[*A parte*] Annotated code and deductive verification

- Provides absolute guarantee
- Automates proofs
- Industrial successes
- Verify wide-spread languages (C, Java, caml . . .)

Three main ingredients:

- operational semantics
- specification language
- proof engine

State of affairs in quantum computing

Three main ingredients:

- operational semantics: matrices → matrix product,
  from Heisenberg (1925), Dirac (1939)
  path-sums (2018)
- specification language: ???
- proof engine: ???

Our strategy

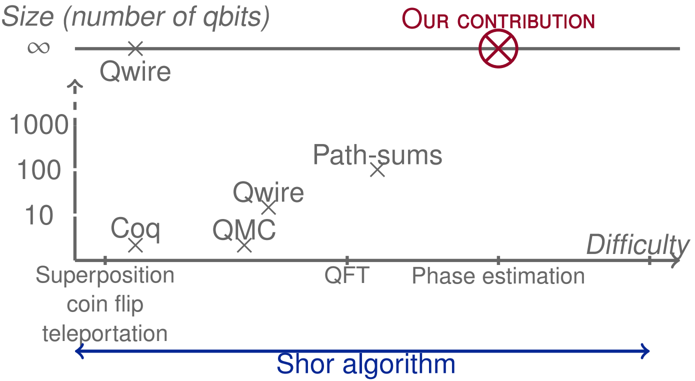
- Build on best practice of formal verification for the classical case
  - separation of concerns
  - scale invariant verification
  - proof automation
  - domain-based specialization
  - flexible specification language
- Tailor them to the quantum case
  - dual semantics (truth reference + specifications)
  - specific reasoning rules
  - dedicated lemmas (1000+) libraries

State of the art

	QMC	Coq	Qwire (Coq)	Path-sums	*Qbricks*
• Separate specification from code	✓	✗	✓	✗	✓
• Scale invariance	✗	✓	✓	✗	✓
• Specifications fitting algorithm	✗	✗	✗	✓	✓
• Automate proofs	✓	✗	✗	✓	↗

Table: Formal verification of quantum circuits

State of the art, achievements in quantum formal verification



Outline

The quantum case : Back to basics

1.	$\vert 0\rangle\vert u\rangle$	initial state
2.	$\rightarrow \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}\vert j\rangle\vert u\rangle$	create superposition
3.	$\rightarrow \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}\vert j\rangle U^j\vert u\rangle$	apply black box
	$= \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}e^{2\pi i j\varphi_u}\vert j\rangle\vert u\rangle$	result of black box
4.	$\rightarrow \vert\widetilde{\varphi_u}\rangle\vert u\rangle$	apply inverse Fourier transform
5.	$\rightarrow \widetilde{\varphi_u}$	measure first register

Algorithm for the quantum phase estimation

The quantum case : Back to basics

1.	$\|0\rangle\|u\rangle$	initial state
2.	$\rightarrow \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}\|j\rangle\|u\rangle$	create superposition
3.	$\rightarrow \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}\|j\rangle U^j\|u\rangle$	apply black box
	$= \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1} e^{2\pi i j\varphi_u}\|j\rangle\|u\rangle$	result of black box
4.	$\rightarrow \|\widetilde{\varphi_u}\rangle\|u\rangle$	apply inverse Fourier transform
5.	$\rightarrow \widetilde{\varphi_u}$	measure first register

Algorithm for the quantum phase estimation

- A sequence of operations
- Intermediate assertions, describing the state of the system at each step

The quantum case : Back to basics

1.	$\|0\rangle\|u\rangle$	initial state
2.	$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} \|j\rangle\|u\rangle$	create superposition
3.	$\rightarrow \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} \|j\rangle U^j\|u\rangle$	apply black box
	$= \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} e^{2\pi i j \varphi_u}\|j\rangle\|u\rangle$	result of black box
4.	$\rightarrow \|\widetilde{\varphi_u}\rangle\|u\rangle$	apply inverse Fourier transform
5.	$\rightarrow \widetilde{\varphi_u}$	measure first register

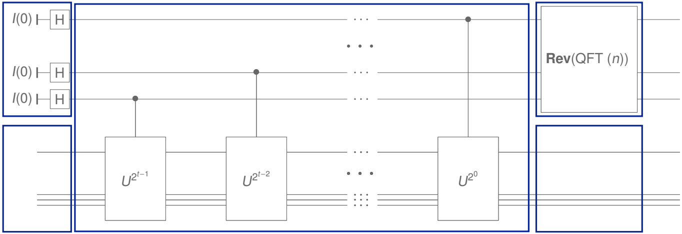
Algorithm for the quantum phase estimation

Derive function specifications, eg :

let create_superposition (state)
pre: $|u\rangle$ is a ket vector
pre: state = $|0\rangle|u\rangle$
post: state = $\frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|u\rangle$
= (* The program *)

- Deductive verification (of annotated programs)
- $\rightarrow$ a domain specific language, embedded in the Why3 environment
  - *build on best practice for the classical case* (automation, separation of concerns, . . .)
  - + interface with SMT-solvers

Circuit building functions




```
type quantum_circuit_pre =
  Phase real | Rx real | Ry real | Rz_ real | Cnot
  | Sequence quantum_circuit_pre quantum_circuit_pre
  | Parallel quantum_circuit_pre quantum_circuit_pre
```

Specification and verification

- Decorate *Qbricks* code with specifications
- Interprete circuit as functions transforming quantum states

$$\begin{array}{l}\text{x: quantum\_state}\\ \text{C: quantum\_circuit}\end{array} \xrightarrow{\text{semantics}} [\![C, x]\!]\text{: quantum\_state}$$

- Path-sum semantics, general form

$$C, |k\rangle_n \xrightarrow[\text{path\_sum\_sem}]{} \frac{1}{\sqrt{2^r}} \sum_{j=0}^{2^r-1} ph(k,j)|ket(i,j)\rangle_n$$

- Three separated parameters, whith recursive definitions:
  - r: int
  - ph : int $\rightarrow$ int $\rightarrow$ complex
  - ket : int $\rightarrow$ int $\rightarrow$ int

Specified circuit building

- Three separated parameters:
  - r: int
  - ph : int → int → complex
  - ket : int → int → int
- functions r (sum_range), ph (phase_part) and ket (ket_part) are defined by recursion for circuits,

```
let rec  function ket_part  (c:quantum_circuit) (bvx bvy: int -> int) (i: int)
  =  match to_pre c with
              Phase_ _ -> bvx i
             | Rx_ _ -> bvy i
             | Ry_ _ -> bvy i
             | Rz_ _ -> bvx i
             | Cnot ->   if  i = 1 then mod (bvx 0 +  bvx 1) 2  else  bvx i
             | Sequence_ d e ->
               ket_part (to_qc e) (ket_part (to_qc d)  bvx bvy)  (shift bvy (sum_range (to_qc d))) i
             | Parallel_  d e -> (concat_fun (ket_part (to_qc d) bvx bvy)
                                  (ket_part (to_qc e)  (shift bvx (depth (to_qc d)))
                                     (shift bvy (sum_range (to_qc d)))) (depth (to_qc d))) i

let rec  function phase_part  (c:quantum_circuit) (bvx bvy: int -> int) ...

let rec  function sum_range  (c:quantum_circuit) (bvx bvy: int -> int) ...

let  function path_sum_sem (c:quantum_circuit) (x: matrix t)
                 ...
  =
(pow_inv_sqrt_2 (sum_range c))  *.. mat_sum (n_bvs (sum_range c))
                                (fun bvy -> phase_part  (ket_to_bv x) bvy*..  (bv_to_ket (ket_part  (ket_to_bv x) bvy)))
```

Specified circuit building

- Three separated parameters:
  - r: int
  - ph : int → int → complex
  - ket : int → int → int
- functions r (sum_range), ph (phase_part) and ket (ket_part) are defined by recursion for circuits,
- they specify circuit *lifted constructors*

```
let function sequence (d e: quantum_circuit): quantum_circuit
      requires{size d = size e}
      ensures{sum_range result = sum_range d + sum_range e}
      ensures{size result = size d}
      ensures{forall bvx bvy: int->int.  forall i :int.
             ket_part result bvx bvy i = if 0<= i < size result then
             ket_part e (ket_part d  bvx bvy) (shift bvy (sum_range d)) i else 0}
      ensures{forall bvx bvy: int->int. phase_part result bvx bvy =
             (phase_part d bvx bvy) *.
             (phase_part e (ket_part d  bvx  bvy) (shift bvy (sum_range d)) )}
  = {to_pre = Sequence (to_pre d) (to_pre e)}
```

Specified circuit building

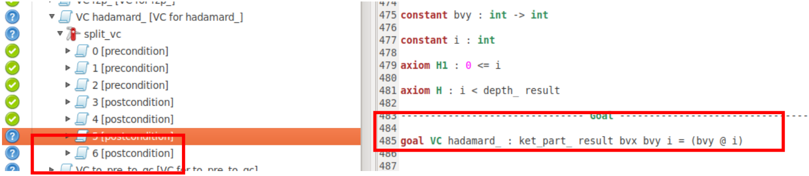
- Three separated parameters:
  - r: int
  - ph : int → int → complex
  - ket : int → int → int
- functions r (sum_range), ph (phase_part) and ket (ket_part) are defined by recursion for circuits,
- they specify circuit *lifted constructors*
- and the circuit building functions

```
let function  hadamard(): quantum_circuit
    ensures{size result = 1}
    ensures{sum_range result = 1}
    ensures{forall bvx bvy: int-> int. forall i :int.
          0<= i < size result -> ket_part result bvx bvy i = bvy i}
    ensures{forall bvx bvy:  int-> int. binary bvx -> binary bvy ->
          phase_part result bvx bvy = value  (dyadic (bvx 0 *  bvy 0) 1) }
  =
  sequence  (rzp 1) (ry (dyadic 1 3))
```

Generating proof obligations (why3)

- Compilation generates proof obligations

```
let function  hadamard(): quantum_circuit
    ensures{size result = 1}
    ensures{sum_range result = 1}
    ensures{forall bvx bvy: int-> int. forall i :int.
           0<= i < size result -> ket_part result bvx bvy i = bvy i}
    ensures{forall bvx bvy:  int-> int. binary bvx -> binary bvy ->
           phase_part result bvx bvy = value  (dyadic (bvx 0 *  bvy 0) 1) }
=
  sequence  (rzp 1) (ry (dyadic 1 3))
```



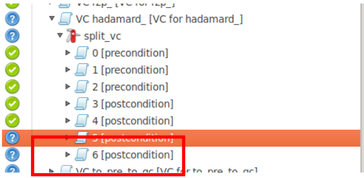
Generating proof obligations (why3)

- Compilation generates proof obligations
- Calling a function provides its postconditions as axioms

```
let function  hadamard(): quantum_circuit
    ensures{size result = 1}
    ensures{sum_range result = 1}
    ensures{forall bvx bvy: int-> int. forall i
           0<= i < size result -> ket_part resu
    ensures{forall bvx bvy:  int-> int. binary
           phase_part result bvx bvy = value
  =
   sequence  (rzp 1) (ry (dyadic 1 3))
```

```
axiom H7 :
  forall bvx1:int -> int, bvy1:int -> int.
   forall i1:int.
    ket_part_ result bvx1 bvy1 i1
    = (if 0 <= i1 /\ i1 < depth_ result
       then ket_part_ (ry (dyadic 1 3))
            ((((fun (y0:quantum_circuit) (y1:int -> int) (y2:int -> int) (y3:
                 int) -> ket_part_ y0 y1 y2 y3)
               @ rzp_ 1)
              @ bvx1)
             @ bvy1)
            (((fun (y0:int -> int) (y1:int) (y2:int) -> shift y0 y1 y2)
              @ bvy1)
             @ sum_range_ (rzp_ 1))
            i1
       else 0)
```

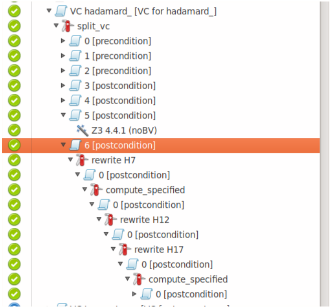
. . .




```
constant bvy : int -> int

constant i : int

axiom H1 : 0 <= i

axiom H : i < depth_ result

---------------------------- Goal ----------------------------

goal VC hadamard_ : ket_part_ result bvx bvy i = (bvy @ i)
```

Supporting proof obligations

- Proof obligations may be sent to SMT-solvers,
- and they can be eased, if needed, by to interactive transformations



Outline

Context

The case for verification of quantum algorithms

Qbricks

- Circuit language
- Dual semantics
- Derive proof obligations
- Toward further automation

Case study: phase estimation algorithm

Conclusion

Path-sum semantics nice properties

- Path-sum semantics satisfies nice properties

Lemma (Properties of function **Ps**)

- ***Linear decomposition.*** *Let c be a quantum circuit, then for any ket* $|u\rangle$ *of length* $\mathbf{s}_c$,

$$\mathbf{Ps}\,(c, |u\rangle) = \sum_{k=0}^{2^{\mathbf{s}_c - 1}} (u(k))\mathbf{Ps}\,(c, |k\rangle_{\mathbf{s}_c})$$

- ***Compositions.*** *Let c and c′ be quantum circuits, let* $|u\rangle$ *and* $|u'\rangle$ *be ket of respective lengths* $\mathbf{s}_c$ *and* $\mathbf{s}_{c'}$. *Then,*
  - $\mathbf{Ps}\,(parallel(c, c'), |u\rangle \otimes |u'\rangle) = (\mathbf{Ps}\,(c, |u\rangle) \otimes (\mathbf{Ps}\,(c', |u'\rangle))$
  - *if c and c′ have the same size then* $\mathbf{Ps}\,(sequence(c, c'), |u\rangle) = \mathbf{Ps}\,(c', \mathbf{Ps}\,(c, |u\rangle))$

- They enable local reasoning without reference to r, ph and ket

Abstract specification: the *eigen* example

Definition (Eigen predicate)

Let c be a quantum circuit, then for any ket $|u\rangle$ of length $\mathbf{s}_c$ and for any complex number v, we say that $|u\rangle$ *is an eigenvector* for c with associated eigenvalue v, and we write $\text{eigen}(c, |u\rangle, v)$, iff

$$\mathbf{Ps}\ (c, |u\rangle) = v|u\rangle$$

Lemma (Eigen sequence composition)

Let c and c' be quantum circuits such that $\mathbf{s}_c = \mathbf{s}_{c'}$, let $|u\rangle$ be a ket of length $\mathbf{s}_c$. Then for any complex values v, v' such that

- $eigen(c, |u\rangle, v)$
- $eigen(c', |u\rangle, v')$

then

$$eigen(sequence(c, c'), |u\rangle, vv')$$

Abstract specification and quantum algorithms

- Quantum algorithms (phase estimation, Grover, quantum simulation, etc.) are often parametrized by an *oracle* quantum circuit, respecting a given property
- This specification may not (nicely) translate in terms of r, ph and ket
- → abstract specifications
- Example, the phase estimation algorithm:

  Input: an unitary operator U and an eigenstate $|v\rangle$ of U

  Output: the eigenvalue associated to $|v\rangle$
- Also: reversed circuit specification, controlled operations specifications, bit permutation specifications, etc.

Specific language fragments and simplified path-semantics

- Predicates may characterize *Qbricks* fragments with simplified path-sum semantics:

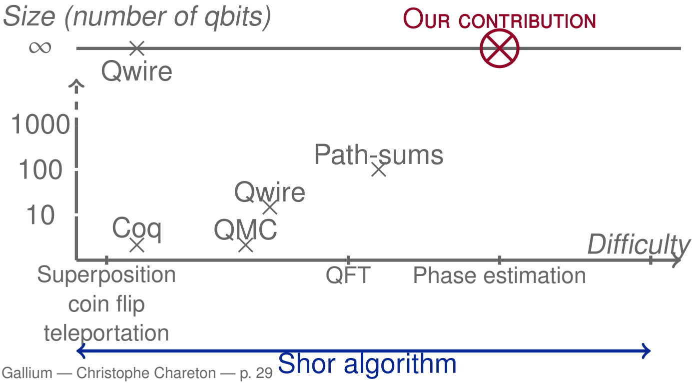
Spec.	generating syntax	Semantics	Design input
$\top$	{Rx, Ry, Rz, Ph, Cnot}	r: int ph : int → int → complex ket : int → int → int	
flat	{Rz, Ph, Cnot}	ph : int → complex ket : int → int	easy specification
diag	{Rz, Ph}	ph : int → complex	easier specification iterators

Outline

Phase estimation

Input: an unitary operator U and an eigenstate $|v\rangle$ of U
Output: the eigenvalue associated to $|v\rangle$

- Eigen decomposition
- Solving linear systems
- Shor (with arithmetic assumption and probability)



Implementation data

	#Lines	#Def.	#Lem	#POs	#Aut.	#Cmd
create_superposition	42	2	1	11	6	36
apply_black_box	57	3	1	50	44	46
QFT	75	3	0	57	51	30
phase estimation	63	4	0	72	65	51
Total	**237**	12	2	**190**	**166**	163

#Aut.: automatically proven POs — #Cmd: interactive commands

Table: Implementation & verification of phase estimation

- The objective is reached: prove by fact that parametrized formal verification for quantum programs is possible
- Future works : further automate proof fulfillment

Comparison of several approaches, QFT algorithm

	QMC	Coq	Qwire (Coq)	Path-sums	*Qbricks*
• Separate specification from code	✓	✗	✓	✗	✓
• Scale invariance	✗	✓	✓	✗	✓
• Specifications fitting algorithm	✗	✗	✗	✓	✓
• Automate proofs	✓	✗	✗	✓	↗

Table: Formal verification of quantum circuits

	#Lines	#Def.	#Lem	#POs	#Aut.	#Cmd
QFT (full *Qbricks*)	**75**	3	0	57	51	**30**
QFT (Path-sum only)	**87**	3	0	73	64	**49**
QFT (Matrix only, cf QMC, Coq solutions)	**200**	8	15	306	285	**106**

Table: Comparison of several approaches, QFT algorithm

Conclusion

- *Qbricks*: a core development framework for certified quantum programming
  - scale invariant
  - close to quantum algorithm descriptions
  - well distinguished from code itself
  - largely automated
- Implementation
  - Circuit building language
  - Dual semantics + equivalence proof
  - Shorcuts for further automation
  - Certified implementation of the phase estimation algorithm
- Future works:
  - Further automate proof framework
  - Extend *Qbricks* to measure → Shor

Commissariat à l'énergie atomique et aux énergies alternatives
CEA Tech List
Centre de Saclay — 91191 Gif-sur-Yvette Cedex
www-list.cea.fr

Etablissement public à caractère industriel et commercial — RCS Paris B 775 685 019