GT IQ

November 28-29th 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

  $\rightarrow$ How to write correct 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

The case for verification of quantum algorithms

Qbricks

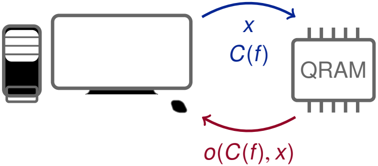
- Circuit language
- Dual semantics
- Derive proof obligations
- Toward further automation

Case study: phase estimation algorithm

Conclusion

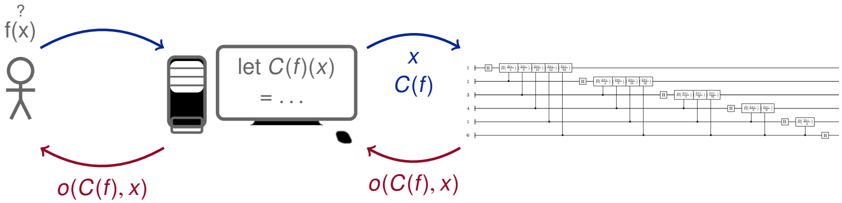
The QRAM model

- 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
- $\rightarrow$ Structured sequences of instructions: quantum circuits



Does the circuit fit the computation need?

How do we check them?

implements

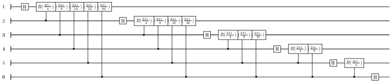
$\|0\rangle\|u\rangle$	initial state
$\rightarrow \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}\|j\rangle\|u\rangle$	create superposition
$\rightarrow \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1}\|j\rangle U^j\|u\rangle$	apply black box
$= \frac{1}{\sqrt{2^t}}\sum_{j=0}^{2^t-1} e^{2\pi i j \varphi_u}\|j\rangle\|u\rangle$	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



How do we check them?

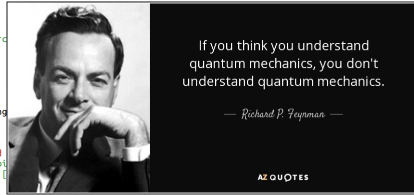
implements

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

Quantum phase estimation (from Nielsen & Chuang)

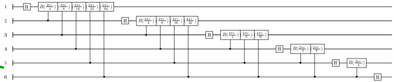
```
qft_internal :: [Qubit] -> Circ
qft_internal [] = return []
qft_internal [x] = do
  hadamard x
  return [x]
qft_internal (x:xs) = do
  xs' <- qft_internal xs
  xs'' <- rotations x xs' (leng
  x' <- hadamard x
  return (x':xs'')
  where
    -- Auxiliary function used
    rotations :: Qubit -> [Qubi
    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 ?



- 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

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

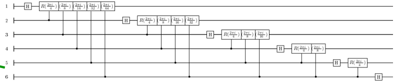
Quantum phase estimation (from Nielsen & Chuang)

```
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)
- specification language: ???
- proof engine: ???

Our goal

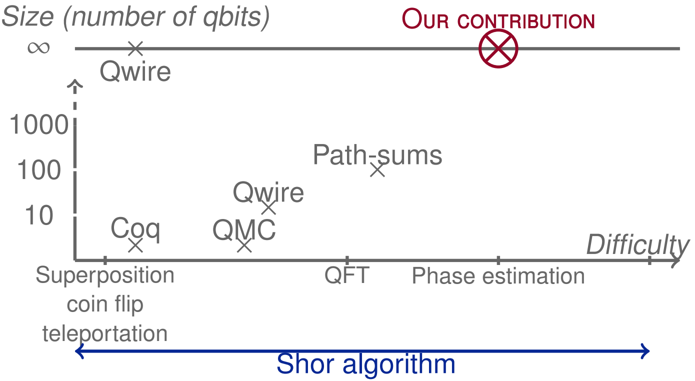
- Specifications for a quantum specification language
  - Specifications fitting algorithm
  - Separate specifications from definitions
    - Easier to adopt
    - Separation of concerns
  - Scale invariance
  - Automate proofs

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 case for verification of quantum algorithms

Qbricks

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

Case study: phase estimation algorithm

Conclusion

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

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

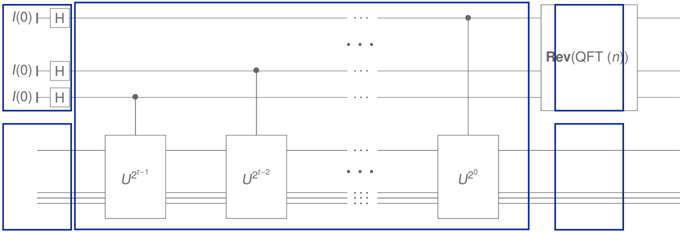
Algorithm for the quantum phase estimation

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 *)

- Functions decorated with pre and post conditions : annotated programming
- $\rightarrow$ embedding in the Why3 environment

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

Leading idea

x: quantum_state
C: quantum_circuit $\xrightarrow{\text{semantics}}$ $[\![C, x]\!]$: 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 → int → complex
- ket : int → int → 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{depth_ d = depth_ e}
      ensures{sum_range_ result = sum_range_ d + sum_range_ e}
      ensures{depth_ result = depth_ d}
      ensures{forall bvx bvy: int->int.  forall i :int.
             0<= i < depth_ result ->  ket_part_ result bvx bvy i =
             ket_part_ e (ket_part_ d  bvx bvy) (shift bvy (sum_range_ d)) i}
      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

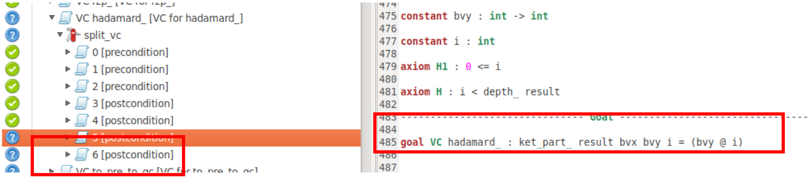
- Three separated parameters:
  - r: int
  - ph : int → int → complex
  - ket : int → int → int
- functions r (sum_range), ph (phase_part) and ket (ket_part) are defined by recursion for circuits,
- they specify circuit *lifted constructors*
- and the circuit building functions

```
let function  hadamard_(): quantum_circuit
    ensures{depth_ result = 1}
    ensures{sum_range_ result = 1}
    ensures{forall bvx bvy: int-> int. forall i :int.
          0<= i < depth_ 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{depth_ result = 1}
    ensures{sum_range_ result = 1}
    ensures{forall bvx bvy: int-> int. forall i :int.
         0<= i < depth_ 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{depth_ result = 1}
    ensures{sum_range_ result = 1}
    ensures{forall bvx bvy: int-> int. forall i
           0<= i < depth_ result -> ket_part_ re
    ensures{forall bvx bvy:  int-> int. binary b
           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)
```

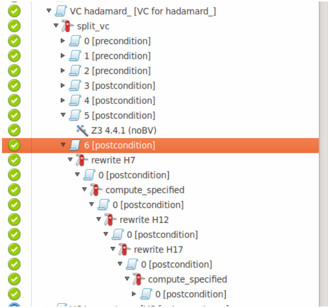
. . .

- VC hadamard_ [VC for hadamard_]
  - split_vc
    - 0 [precondition]
    - 1 [precondition]
    - 2 [precondition]
    - 3 [postcondition]
    - 4 [postcondition]
    - 5 [postcondition]
    - 6 [postcondition]
- VC to_pre_to_gc [VC for to_pre_to_gc]

```
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



Toward further automation

- reasoning abstractly from path-sums (instead of r, ph and ket)
  - Nice path-sum theorems:
    - Linearity
    - Translate sequence as function composition
    - Translate parallel as Kronecker product
  - Enables abstract specifications:
    - Eigen value specifications
    - Controlled operations, etc
  - Precious when dealing with underspecified circuit parameters
- Simplified path-semantics, for adequate language fragments

Property	Class of circuits	Design input
flat	{rz, ph, cnot} syntax	easy specification
diag	{rz, ph} syntax	very easy specification iterators

Outline

The case for verification of quantum algorithms

Qbricks

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

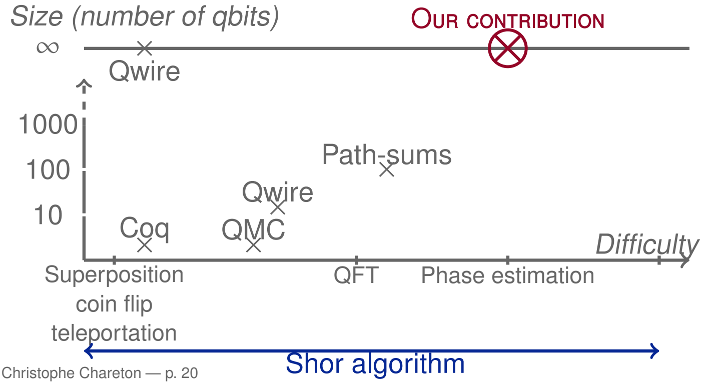
Case study: phase estimation algorithm

Conclusion

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)



Case study

	#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

	#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)	200	8	15	306	285	106

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

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