Verification in Quantum Computing

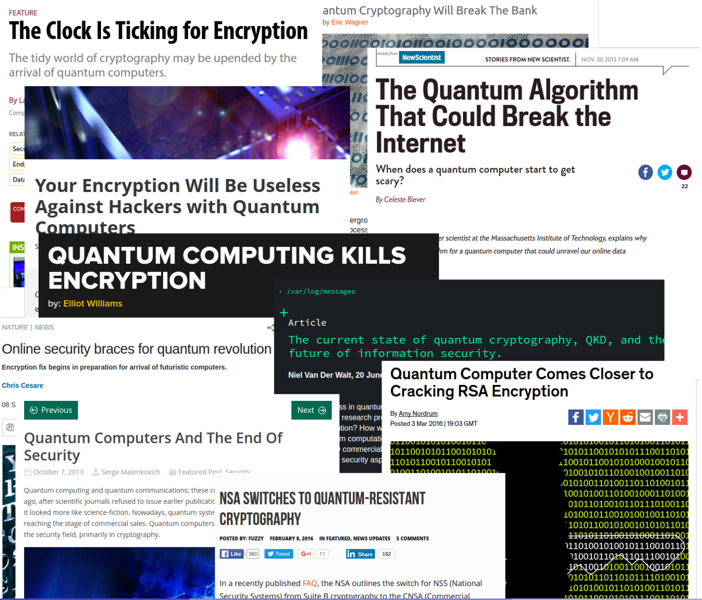
Matthew Amy

University of Waterloo & Institute for Quantum Computing

Design Automation for Quantum Computing
November 16th, 2017

Quantum computing

Theory:



Quantum computing

Reality:

Quantum computing is weakened by a high degree of overhead

Sources of overhead:

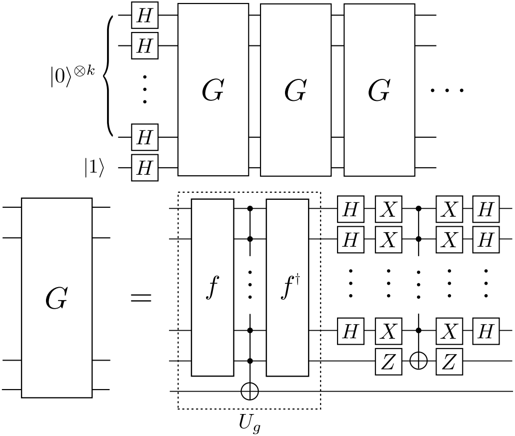
- Intrinsic overhead of an algorithm

  e.g. overhead of Grover's search
- Overhead incurred at the logical layer due to reversibility

  e.g. $g : |x\rangle|y\rangle \rightarrow |x\rangle|y \oplus f(x)\rangle$
- Additional overhead at the physical layer due to error correction

Example

Breaking SHA (arXiv:1603.09383)



Algorithmic overhead: Additional query of f, $4n - 8$ Toffolis
Logical overhead: 1600 qubits, $> 2\times$ the number of gates Physical overhead: 2^{38} times as many "executions of SHA-256"

Resource estimation

Estimate how much resources (time & space) a realistic implementation of an algorithm uses

Typical design flow (e.g. Quipper, QCL):

High-level code with irreversible functions as oracles

↓

Compile oracles into reversible circuits

↓

Optimize circuit(s)

↓

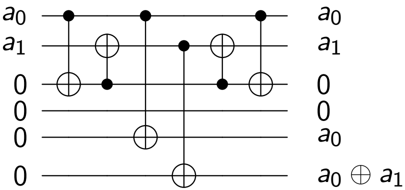
Use compiled circuit metrics to estimate error correction

Errors can (and do) occur at any stage!

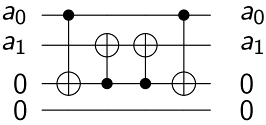
Example

Eager cleanup bug

Without optimization:



With optimization:



Why verify?

1.) *Quantum resource estimates are being used to guide real security policies*

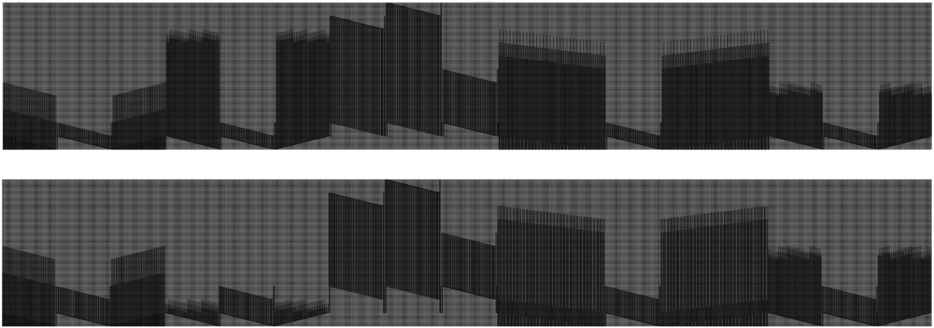
- Open Quantum Safe (https://openquantumsafe.org/)
- Bitcoin (Aggarwal et. al. arXiv:1710.10377)
- Symmetric key systems (Ling et. al. arXiv:1707.02766)
- Resource analyses of AES (Grassl et. al. arxiv:1512.04965), SHA (Amy et. al. arXiv:1603.09383) etc.

2.) *Resource estimates vary wildly between compilers*

e.g. for binary welded tree ($n = 100$,$s = 100$)

- ScaffCC gives 571805 qubits, 33966707 gates
- Quipper gives 314/1932 qubits, 30424410/36257210 gates

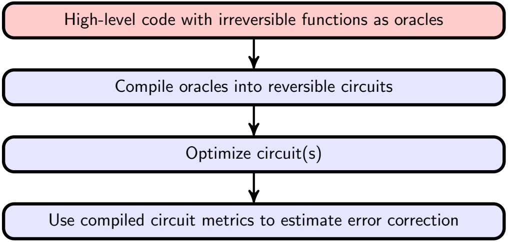
Why verify formally?



3.) Testing capability is limited

- Quantum simulation doesn't scale
- Circuits are special-purpose and monolithic

Verifying a resource analysis design flow



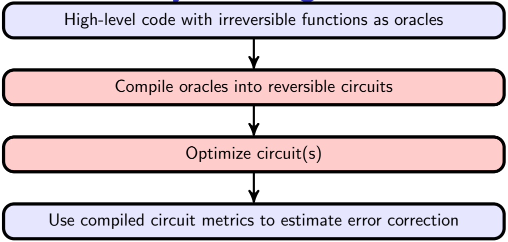
Program verification

- Prove properties of expected behaviour for specific programs
- Properties may not be true of all programs, e.g. integer overflow
- Techniques include abstract interpretation (Entanglement analysis), model checking (Quantum model-checker), type systems (Quipper), formal proof (Quantum Hoare Logic)

Quantum-specific challenges:

- What are the program properties of interest?

Verifying a resource analysis design flow



Compiler verification

- Compiled program executes as expected
- Techniques include translation validation (per program), formal proof (all programs)
- CompCert, CAKEML A Verified Implementation of ML, ReVerC

Quantum-specific challenges:

- Explicit clean-up and reuse of memory
- Probabilistic semantics

Formal proof in compiler verification

ML-like language with dependent types developed at MSR

What are Dependent types?

- Types may depend on terms – i.e. Array n
- Corresponds to predicate logic (Curry-Howard isomorphism)

What are they useful for? writing logical specifications/theorems

```
val head : l:List{not (is_Empty l)} -> Tot int
val insert_is_heap : h:Heap -> i:int ->
  Lemma (is_heap h ⇒ is_heap (insert h i))
val compile_correct :
  Lemma (∀ P:program, i:inputs.
    eval_program P i = eval_assembly (compile P) i)
```

How do we verify specifications/theorems are correct?

- F* compiler checks specifications with SMT solver

caveat: typically have to write lemmas & induction structure

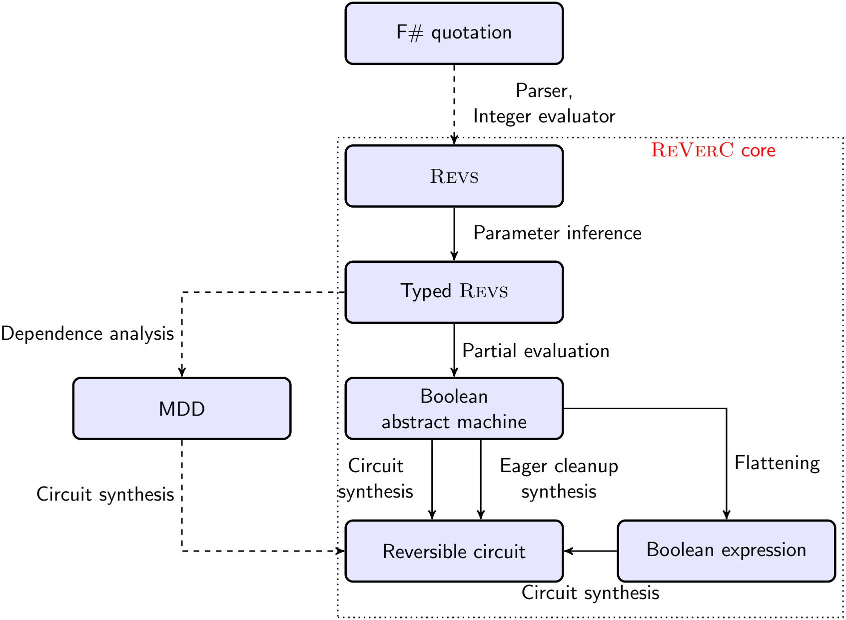
ReVerC (arXiv:1603.01635)

https://github.com/msr-quarc/ReVerC

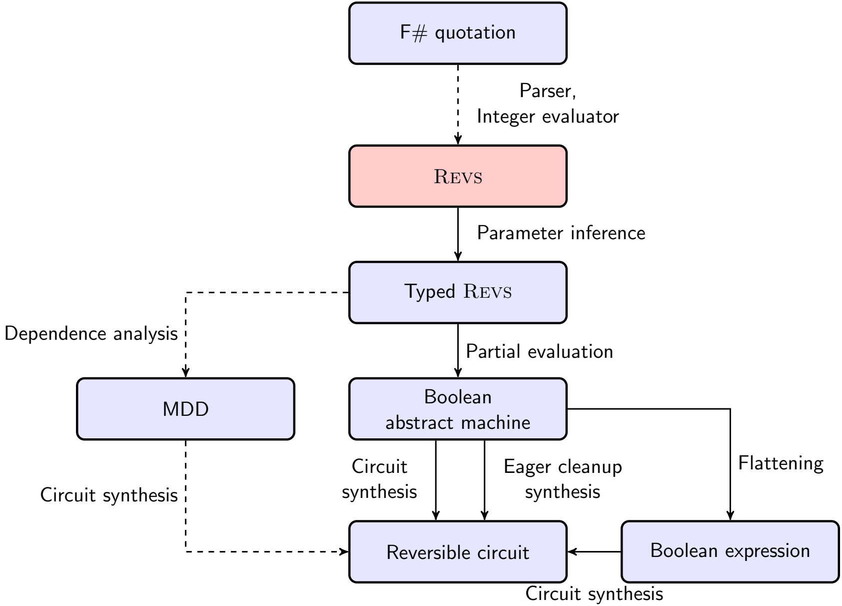
Reversible circuit compiler for the F# embedded DSL Revs

- Compiles irreversible code into reversible circuits
- Performs optimizations for space-efficiency
- Formally verified in F*
- Includes a BDD-based assertion-checker for program verification & additional translation validation

Compiler architecture



Revs


F# quotation
Parser,
Integer evaluator
Revs
Parameter inference
Typed Revs
Dependence analysis
Partial evaluation
MDD
Boolean
abstract machine
Circuit
synthesis
Eager cleanup
synthesis
Flattening
Circuit synthesis
Reversible circuit
Boolean expression
Circuit synthesis

Revs

$$
\begin{aligned}
&\textbf{Var } x, \quad \textbf{Bool } b \in \{0,1\} = \mathbb{B}, \quad \textbf{Nat } i,j \in \mathbb{N}, \quad \textbf{Loc } l \in \mathbb{N} \\
&\textbf{Val } v ::= \text{unit} \mid l \mid \text{reg } l_1 \ldots l_n \mid \lambda x.t \\
&\textbf{Term } t ::= \text{let } x = t_1 \text{ in } t_2 \mid \lambda x.t \\
&\qquad \mid (t_1\ t_2) \\
&\qquad \mid t_1 ; t_2 \\
&\qquad \mid x \\
&\qquad \mid t_1 \leftarrow t_2 \\
&\qquad \mid b \mid t_1 \oplus t_2 \mid t_1 \wedge t_2 \\
&\qquad \mid \text{reg } t_1 \ldots t_n \mid t.[i] \mid t.[i..j] \mid \text{append } t_1\ t_2 \mid \text{rotate } i\ t \\
&\qquad \mid \text{clean } t \mid \text{assert } t
\end{aligned}
$$

REVS by example

n-bit adder

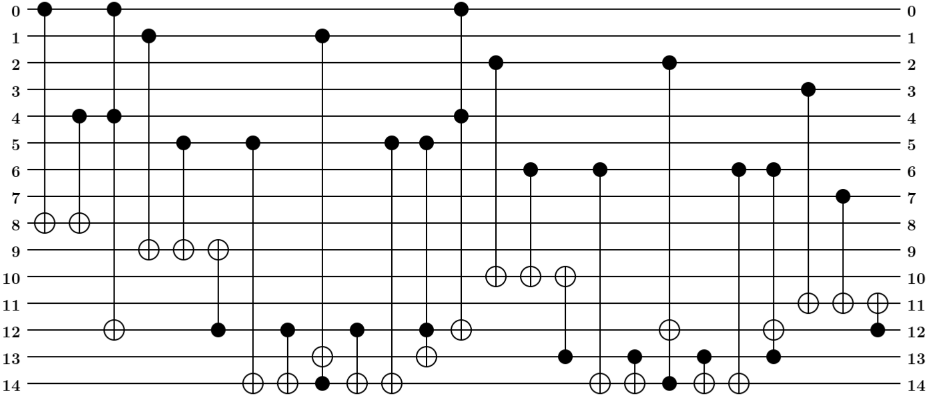
```
let adder n = <@
  fun a b ->
    let maj a b c = (a ∧ (b ⊕ c)) ⊕ (b ∧ c)
    let result = Array.zeroCreate(n)
    let mutable carry = false

    result.[0] ← a.[0] ⊕ b.[0]
    for i in 1 .. n-1 do
      carry ← maj a.[i-1] b.[i-1] carry
      result.[i] ← a.[i] ⊕ b.[i] ⊕ carry
      assert result.[i] = (a.[i] ⊕ b.[i] ⊕ carry)
    result
@>
```

**Note: all control is compile-time static

REVS by example

n-bit adder



Revs by example

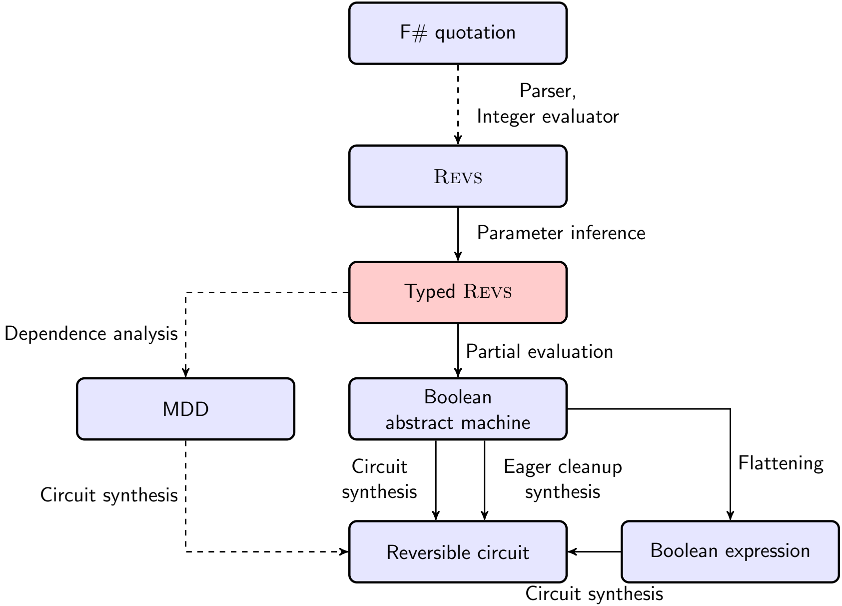
SHA-256

```
let s0 a =
 let a2 = rot 2 a
 let a13 = rot 13 a
 let a22 = rot 22 a
 let t = Array.zeroCreate 32
 for i in 0 .. 31 do
   t.[i] ← a2.[i] ⊕
          a13.[i] ⊕
          a22.[i]
 t
let s1 a =
 let a6 = rot 6 a
 let a11 = rot 11 a
 let a25 = rot 25 a
 let t = Array.zeroCreate 32
 for i in 0 .. 31 do
   t.[i] ← a6.[i] ⊕
          a11.[i] ⊕
          a25.[i]
 t
let ma a b c =
 let t = Array.zeroCreate 32
 for i in 0 .. 31 do
   t.[i] ← (b.[i] ∧ c.[i]) ⊕
     (a.[i] ∧ (b.[i] ⊕ c.[i]))
 t
```

```
let ch e f g =
 let t = Array.zeroCreate 32
 for i in 0 .. 31 do
   t.[i] ← e.[i] ∧ f.[i] ∧ g.[i]
 t

fun k w x →
 let hash x =
   let a = x.[0..31],
     b = x.[32..63]
     c = x.[64..95],
     d = x.[96..127],
     e = x.[128..159],
     f = x.[160..191],
     g = x.[192..223],
     h = x.[224..255]
   (%modAdd 32) (ch e f g) h
   (%modAdd 32) (s0 a) h
   (%modAdd 32) w h
   (%modAdd 32) k h
   (%modAdd 32) h d
   (%modAdd 32) (ma a b c) h
   (%modAdd 32) (s1 e) h
 for i in 0 .. n - 1 do
   hash (rot 32*i x)
 x
```

Typed REVS


F# quotation
Parser,
Integer evaluator
REVS
Parameter inference
Typed REVS
Dependence analysis
Partial evaluation
MDD
Boolean
abstract machine
Circuit synthesis
Circuit
synthesis
Eager cleanup
synthesis
Flattening
Reversible circuit
Boolean expression
Circuit synthesis

Typed REVS

Type $T ::= X \mid \text{Unit} \mid \text{Bool} \mid \text{Reg } n \mid T_1 \to T_2$

Inferred type system with statically typed registers sizes

- Main purpose is to simplify the job of the compiler
  - Simpler compiler ⇒ easier verification!
- Verification-light
  - Prevents out-of-bounds register accesses
  - Sanity check for register sizes

```
let f = fun a : Reg 8 -> ... in
let a = Array.zeroCreate 8 in
let b = Array.zeroCreate 16 in
f a
f b
```

Type/parameter inference

Basic idea: solve a system of integer linear arithmetic constraints

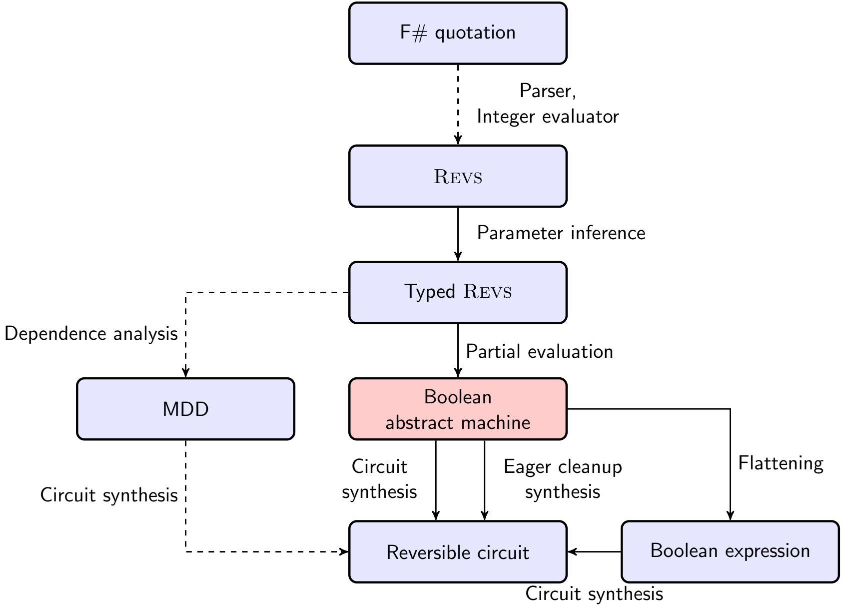
- e.g. $(x = \text{ Reg } y) \wedge (y \geq z - 3) \wedge (y \geq 8)$
- let c = append a b $\rightarrow$
  $(c : \text{Reg } x) \wedge (a : \text{Reg } y) \wedge (b : \text{Reg } z) \wedge (x \geq y + z)$

Solver overview:

- Solve equalities by unification
- Merge arithmetic constraints & reduce to normal form
- For constraints $x \geq n$, set $x = n$
- Check remaining arithmetic constraints are satisfied

Caveat: doesn't always find a solution

Boolean abstract machine



Boolean abstract machine

Only one operation:

assign a store location to the result of a Boolean expression

Partial evaluation used to transform REVS code into the abstract machine

- Lvalue most be a new, 0-valued store location
- RHS is a Boolean expression
- Semantics & transformation coincide $\Rightarrow$ easier verification!

**Strictly more general than reversible circuits

Example

Adder circuit

```
fun a b ->
  let carry_ex a b c = (a ∧ (b ⊕ c)) ⊕ (b ∧ c)
  let result = Array.zeroCreate(4)
  let mutable carry = false

  result.[0] ← a.[0] ⊕ b.[0]
  for i in 1 .. 4-1 do
    carry ← carry_ex a.[i-1] b.[i-1] carry
    result.[i] ← a.[i] ⊕ b.[i] ⊕ carry
    assert (result.[i] = (a.[i] ⊕ b.[i] ⊕ carry))
  result
```

↓ partial evaluation

```
(* result = alloc(4), carry0 = alloc(1) *)
result.[0] ← a.[0] ⊕ b.[0]
carry1     ← (a.[0] ∧ (b.[0] ⊕ carry0)) ⊕ (b.[0] ∧ carry0)
result.[1] ← a.[1] ⊕ b.[1] ⊕ carry1
carry2     ← (a.[1] ∧ (b.[1] ⊕ carry1)) ⊕ (b.[1] ∧ carry1)
result.[2] ← a.[2] ⊕ b.[2] ⊕ carry2
carry3     ← (a.[2] ∧ (b.[2] ⊕ carry2)) ⊕ (b.[2] ∧ carry2)
result.[3] ← a.[3] ⊕ b.[3] ⊕ carry3
```

Recall

Reversible computing

Every operation must be invertible

- $x \wedge y = 0 \implies x = ???, y = ???$
- Can't re-use memory without "uncomputing" its value first

To perform classical functions reversibly, embed in a larger space

- $\mathit{Toffoli}(x, y, z) = (x, y, z \oplus (x \wedge y))$
- $\mathit{Toffoli}(x, y, 0) = (x, y, x \wedge y)$

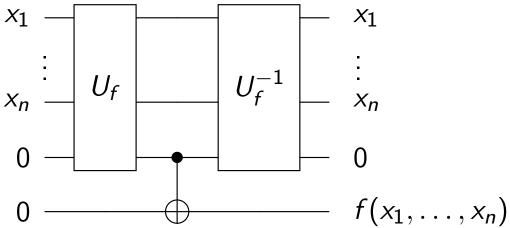
Recall

Reclaiming space

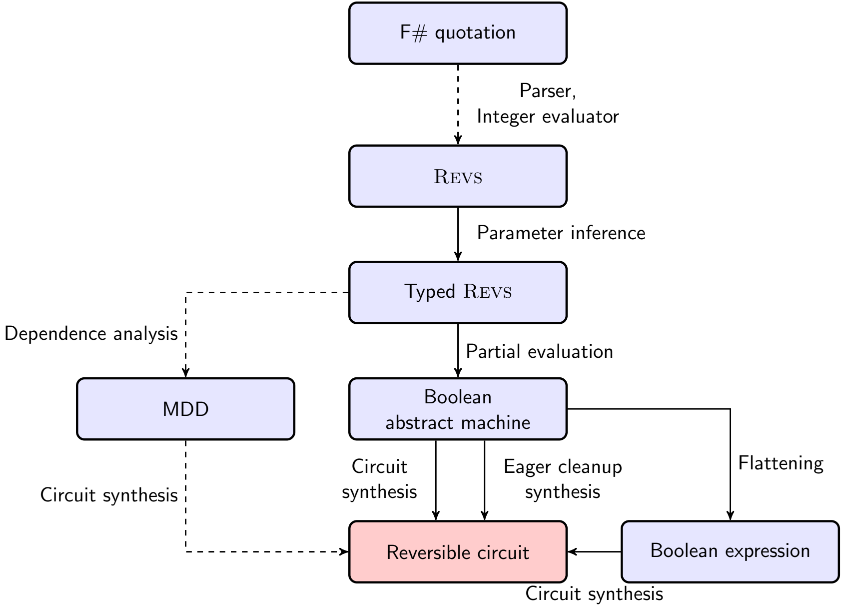
Naïve "reversibilification": replace every AND gate with a Toffoli

- Temporary bits are called ancillas
- Uses space linear(!) in the number of AND gates

Bennett's trick: copy out result of a computation & uncompute



Circuit compilation



Eager Cleanup

A.K.A. garbage collection

```
(* result = alloc(4), carry_0 = alloc(1) *)
1 result.[0] ← a.[0] ⊕ b.[0]
2 carry_1    ← (a.[0] ∧ (b.[0] ⊕ carry_0)) ⊕ (b.[0] ∧ carry_0)
3 result.[1] ← a.[1] ⊕ b.[1] ⊕ carry_1
4 carry_2    ← (a.[1] ∧ (b.[1] ⊕ carry_1)) ⊕ (b.[1] ∧ carry_1)
5 result.[2] ← a.[2] ⊕ b.[2] ⊕ carry_2
6 carry_3    ← (a.[2] ∧ (b.[2] ⊕ carry_2)) ⊕ (b.[2] ∧ carry_2)
7 result.[3] ← a.[3] ⊕ b.[3] ⊕ carry_3
```

After line 4, we can garbage-collect $\texttt{carry}_1$ and reuse its space for $\texttt{carry}_3$

Problem: we can't overwrite $\texttt{carry}_1$ with the 0 state
Solution: each location i is associated with an expression $\kappa(i)$ s.t.

$$i \oplus \kappa(i) = 0$$

Interpretations

Compilation methods defined by providing interpretations $\mathcal{I}$ of the abstract machine

An interpretation consists of a domain D and two operations

$$\text{assign} : D \times \mathbb{N} \times \textbf{BExp} \to D$$
$$\text{eval} : D \times \mathbb{N} \times \textbf{State} \rightharpoonup \mathbb{B}.$$

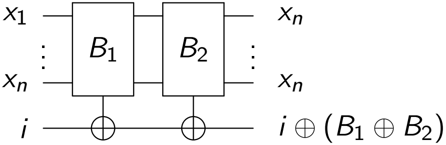
**Semantic function eval is provided to unify verification

Circuit synthesis

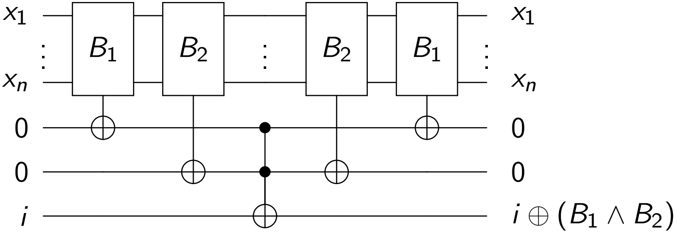
Bexp $B ::= 0 \mid 1 \mid i \mid \neg B \mid B_1 \oplus B_2 \mid B_1 \wedge B_2$

To be reversible compiled expression must have the form $i \oplus B$

$i \oplus (B_1 \oplus B_2) \quad \rightarrow$



$i \oplus (B_1 \wedge B_2) \quad \rightarrow$



Eager Cleanup

A.K.A. garbage collection

In the 4-bit adder example, after the assignment

```
carry₂ ← (a.[1] ∧ (b.[1] ⊕ carry₁)) ⊕ (b.[1] ∧ carry₁)
```

the location of $\texttt{carry}_1$ is no longer in use, so we can reuse it for $\texttt{carry}_3$

Problem: we can't overwrite $\texttt{carry}_1$ with the "0" state

Solution: if $\texttt{carry}_1$ is in the state B, $\texttt{carry}_1 \oplus B = 0$

$\Rightarrow$ location i is associated with an expression $\kappa(i)$ such that $i \oplus \kappa(i) = 0$

Eager Cleanup

```
1 c₁ ← a.[0] ∧ b.[0]
2 c₂ ← (a.[1] ∧ (b.[1] ⊕ c₁)) ⊕ (b.[1] ∧ c₁)
3 clean c₁ (* c₁ ← c₁ ⊕ κ(c₁) *)
4 c₃ ← (a.[2] ∧ (b.[2] ⊕ c₂)) ⊕ (b.[2] ∧ c₂)
5 clean c₂ (* c₂ ← c₂ ⊕ κ(c₂) *)
6
```

l	$\kappa(c_1)$	$\kappa(c_2)$	$\kappa(c_1)$
1	0	0	0
2			
3			
4			
5			
6			

Eager Cleanup

```
1 c1 ← a.[0] ∧ b.[0]
2 c2 ← (a.[1] ∧ (b.[1] ⊕ c1)) ⊕ (b.[1] ∧ c1)
3 clean c1 (* c1 ← c1 ⊕ κ(c1) *)
4 c3 ← (a.[2] ∧ (b.[2] ⊕ c2)) ⊕ (b.[2] ∧ c2)
5 clean c2 (* c2 ← c2 ⊕ κ(c2) *)
6
```

l	$\kappa(c_1)$	$\kappa(c_2)$	$\kappa(c_1)$
1	0	0	0
2	$a_0 \wedge b_0$	0	0
3			
4			
5			
6			

Eager Cleanup

```
1 c₁ ← a.[0] ∧ b.[0]
2 c₂ ← (a.[1] ∧ (b.[1] ⊕ c₁)) ⊕ (b.[1] ∧ c₁)
3 clean c₁ (* c₁ ← c₁ ⊕ κ(c₁) *)
4 c₃ ← (a.[2] ∧ (b.[2] ⊕ c₂)) ⊕ (b.[2] ∧ c₂)
5 clean c₂ (* c₂ ← c₂ ⊕ κ(c₂) *)
6
```

l	$\kappa(c_1)$	$\kappa(c_2)$	$\kappa(c_1)$
1	0	0	0
2	$a_0 \wedge b_0$	0	0
3	$a_0 \wedge b_0$	$(a_1 \wedge (b_1 \oplus c_1)) \oplus (b_1 \wedge c_1)$	0
4			
5			
6			

Eager Cleanup

```
1 c1 ← a.[0] ∧ b.[0]
2 c2 ← (a.[1] ∧ (b.[1] ⊕ c1)) ⊕ (b.[1] ∧ c1)
3 clean c1 (* c1 ← c1 ⊕ κ(c1) *)
4 c3 ← (a.[2] ∧ (b.[2] ⊕ c2)) ⊕ (b.[2] ∧ c2)
5 clean c2 (* c2 ← c2 ⊕ κ(c2) *)
6
```

l	$\kappa(c_1)$	$\kappa(c_2)$	$\kappa(c_1)$
1	0	0	0
2	$a_0 \wedge b_0$	0	0
3	$a_0 \wedge b_0$	$(a_1 \wedge (b_1 \oplus c_1)) \oplus (b_1 \wedge c_1)$	0
4	0	$(a_1 \wedge (b_1 \oplus (a_0 \wedge b_0))) \oplus (b_1 \wedge (a_0 \wedge b_0))$	0
5			
6			

Eager Cleanup

```
1 c₁ ← a.[0] ∧ b.[0]
2 c₂ ← (a.[1] ∧ (b.[1] ⊕ c₁)) ⊕ (b.[1] ∧ c₁)
3 clean c₁ (* c₁ ← c₁ ⊕ κ(c₁) *)
4 c₃ ← (a.[2] ∧ (b.[2] ⊕ c₂)) ⊕ (b.[2] ∧ c₂)
5 clean c₂ (* c₂ ← c₂ ⊕ κ(c₂) *)
6
```

l	$\kappa(c_1)$	$\kappa(c_2)$	$\kappa(c_1)$
1	0	0	0
2	$a_0 \wedge b_0$	0	0
3	$a_0 \wedge b_0$	$(a_1 \wedge (b_1 \oplus c_1)) \oplus (b_1 \wedge c_1)$	0
4	0	$(a_1 \wedge (b_1 \oplus (a_0 \wedge b_0))) \oplus (b_1 \wedge (a_0 \wedge b_0))$	0
5	0	$(a_1 \wedge (b_1 \oplus (a_0 \wedge b_0))) \oplus (b_1 \wedge (a_0 \wedge b_0))$	$(a_2 \wedge (b_2 \oplus c_2)) \oplus (b_2 \wedge c_2)$
6			

Eager Cleanup

```
1 c₁ ← a.[0] ∧ b.[0]
2 c₂ ← (a.[1] ∧ (b.[1] ⊕ c₁)) ⊕ (b.[1] ∧ c₁)
3 clean c₁ (* c₁ ← c₁ ⊕ κ(c₁) *)
4 c₃ ← (a.[2] ∧ (b.[2] ⊕ c₂)) ⊕ (b.[2] ∧ c₂)
5 clean c₂ (* c₂ ← c₂ ⊕ κ(c₂) *)
6
```

l	$\kappa(c_1)$	$\kappa(c_2)$	$\kappa(c_1)$
1	0	0	0
2	$a_0 \wedge b_0$	0	0
3	$a_0 \wedge b_0$	$(a_1 \wedge (b_1 \oplus c_1)) \oplus (b_1 \wedge c_1)$	0
4	0	$(a_1 \wedge (b_1 \oplus (a_0 \wedge b_0))) \oplus (b_1 \wedge (a_0 \wedge b_0))$	0
5	0	$(a_1 \wedge (b_1 \oplus (a_0 \wedge b_0))) \oplus (b_1 \wedge (a_0 \wedge b_0))$	$(a_2 \wedge (b_2 \oplus c_2)) \oplus (b_2 \wedge c_2)$
6	0	0	???

Verification

Formal verification of ReVerC[1] carried out in F*

$\sim$ 2000 lines of code

$\sim$ 2200 lines of proof code, written in 1 "person month"

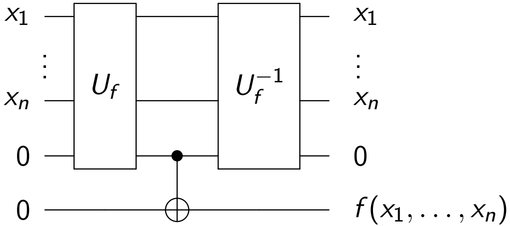
Main theorems:

- Circuit synthesis produces correct output
- Circuit synthesis cleans all intermediate ancillas
- Each abstract machine compiler preserves the semantics
- All optimizations correct, etc.

[1] https://github.com/msr-quarc/ReVerC

Verifying Bennett

The Bennett trick:

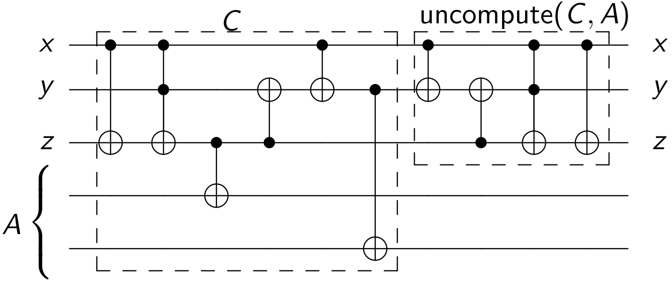


Works because the middle gate does not affect bits used in U_f

Verifying Bennett

A generalized Bennett method

Given a circuit C and set of bits A, we can uncompute C on $\overline{A}$ if no bits of A are used as controls in C



Verifying Bennett

```
val bennett : C:circuit -> copy:circuit -> st:state ->
  Lemma (requires (wfCirc C /\ disjoint (uses C) (mods copy)))
        (ensures  (agree_on st
                     (evalCirc (C@copy@(rev C)) st)
                     (uses C)))
let bennett C copy st =
  let st', st'' = evalCirc C st, evalCirc (C@copy) st in
    eval_mod st' copy;
    ctrls_sub_uses (rev C);
    evalCirc_state_swap (rev C) st' st'' (uses C);
    rev_inverse C st

val uncompute_mixed_inverse : C:circuit -> A:set int -> st:state ->
  Lemma (requires (wfCirc C /\ disjoint A (ctrls C)))
        (ensures  (agree_on st
                     (evalCirc (rev (uncompute C A)) (evalCirc C st))
                     (complement A)))
let uncompute_mixed_inverse C A st =
  uncompute_agree C A st;
  uncompute_ctrls_subset C A;
  evalCirc_state_swap (rev (uncompute C A))
                      (evalCirc C st)
                      (evalCirc (uncompute C A) st)
                      (complement A);
  rev_inverse (uncompute C A) st
```

Verification

```
(* Circuit synthesis correctness *)
val compile_bexp_correct : ah:ancHeap -> targ:int ->
                           exp:boolExp -> st:state ->
  Lemma (requires (zeroHeap st ah /\
                   disjoint (elts ah) (vars exp) /\
                   not (Set.mem targ (elts ah)) /\
                   not (Set.mem targ (vars exp))))
        (ensures  (compileBexpEval ah targ exp st =
                    (lookup st targ) <> evalBexp exp st))
```

Verification

```
(* Circuit synthesis cleans ancillas *)
val compile_with_cleanup : ah:ancHeap -> targ:int ->
                           exp:boolExp -> st:state ->
  Lemma (requires (zeroHeap st ah /\
                   disjoint (elts ah) (vars exp) /\
                   not (Set.mem targ (elts ah)) /\
                   not (Set.mem targ (vars exp))))
        (ensures  (zeroHeap (compileBexpCleanEvalSt ah targ exp st)
                    (first (compileBexpClean ah targ exp))))
```

Verification

```
(* "Circuit" interpretation preserves semantics *)
type valid_circ_state (cs:circState) (init:state) =
  (forall l l'. not (l = l') ==>
    not (lookup cs.subs l = lookup cs.subs l')) /\
  disjoint (vals cs.subs) (elts cs.ah) /\
  zeroHeap init cs.ah /\
  zeroHeap (evalCirc cs.gates init) cs.ah /\
  (forall bit. Set.mem bit (vals cs.subs) ==>
    (lookup cs.zero bit = true ==>
      lookup (evalCirc cs.gates init) bit = false))

type equiv_state (cs:circState) (bs:boolState) (init:state) =
  cs.top = forall i. circEval cs init i = boolEval bs init i

val assign_pres_equiv : cs:circState -> bs:boolState -> l:int ->
                        bexp:boolExp -> init:state ->
  Lemma (requires (valid_circ_state cs init /\ equiv_state cs bs init)
        (ensures  (valid_circ_state (circAssign cs l bexp) init /\
                   equiv_state (circAssign cs l bexp)
                               (boolAssign bs l bexp) init))
```

Verification

```
(* "Eager cleanup" interpretation preserves semantics *)
type valid_GC_state (cs:circGCState) (init:state) =
  (forall l l'. not (l = l') ==>
    not (lookup cs.symtab l = lookup cs.symtab l')) /\
  (disjoint (vals cs.symtab) (elts cs.ah)) /\
  (zeroHeap init cs.ah) /\
  (zeroHeap (evalCirc cs.gates init) cs.ah) /\
  (forall bit. Set.mem bit (vals cs.symtab) ==>
    (disjoint (vars (lookup cs.cvals bit)) (elts cs.ah))) /\
  (forall bit. Set.mem bit (vals cs.symtab) ==>
    (b2t(lookup cs.isanc bit) ==> lookup init bit = false)) /\
  (forall bit. Set.mem bit (vals cs.symtab) ==>
    (evalBexp (BXor (BVar bit, (lookup cs.cvals bit)))
              (evalCirc cs.gates init) = lookup init bit))

type equiv_state (cs:circGCState) (bs:boolState) (init:state) =
  cs.top = forall i. circGCEval cs init i = boolEval bs init i

val assign_pres_equiv : cs:circGCState -> bs:boolState -> l:int ->
                        bexp:boolExp -> init:state ->
  Lemma (requires (valid_GC_state cs init /\ equiv_state cs bs init))
        (ensures  (valid_GC_state (circGCAssign cs l bexp) init /\
                   equiv_state (circGCAssign cs l bexp)
                               (boolAssign bs l bexp) init))
```

Experiments

Bit counts with eager cleanup $\sim$ to state-of-the-art compiler

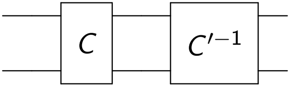
Benchmark	Revs (eager)			ReVerC (eager)		
	bits	gates	Toffolis	bits	gates	Toffolis
carryRippleAdd 32	129	467	124	**113**	361	90
carryRippleAdd 64	257	947	252	**225**	745	186
mult 32	128	6016	4032	128	6016	4032
mult 64	256	24320	16256	256	24320	16256
carryLookahead 32	**109**	1036	344	146	576	146
carryLookahead 64	**271**	3274	1130	376	1649	428
modAdd 32	65	188	62	65	188	62
modAdd 64	129	380	126	129	380	126
cucarroAdder 32	65	98	32	65	98	32
cucarroAdder 64	129	194	64	129	194	64
ma4	17	24	8	17	24	8
SHA-2 round	**353**	2276	754	449	1796	**594**
MD5	7905	82624	27968	**4769**	70912	**27520**

Towards functional verification

Given a circuit C, can we verify that C implements a unitary matrix U? What about an optimized circuit C'?

The reversible case

- Classical CAD techniques such as miters & BDDs or SAT solvers applicable here



- BDD-based verification in ReVerC starts thrashing at $\sim$ 75 bits with 8 Gb memory
- May be able to go further with functional coverage techniques

The quantum case

- Decision diagram-based techniques applied in the past (QuIDD)
- Limited by size of unitaries

Sum-over-paths

A space-efficient, natural mathematical description of unitaries

$$R_z(\theta) : |x\rangle \mapsto e^{2\pi i\theta x}|x\rangle$$

$$H : |x\rangle \mapsto \frac{1}{\sqrt{2}} \sum_{y\in\{0,1\}} \omega^{4xy}|y\rangle$$

$$\text{Toffoli}_n : |x_1x_2\cdots x_n\rangle \mapsto |x_1x_2\cdots(x_1 \wedge x_2 \wedge \cdots \wedge x_n)\rangle$$

$$\text{Adder}_n : |\mathbf{x}\rangle|\mathbf{y}\rangle|0\rangle \mapsto |\mathbf{x}\rangle|\mathbf{y}\rangle|\mathbf{x}+\mathbf{y}\rangle$$

$$\text{QFT}_n : |\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2}^n} \sum_{\mathbf{y}=0}^{2^n-1} e^{2\pi i\mathbf{x}\mathbf{y}/2^n}|\mathbf{y}\rangle$$

In general:

$$U : |\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2}^k} \sum_{\mathbf{y}\in\{0,1\}^k} e^{2\pi i p(\mathbf{x},\mathbf{y})}|f(\mathbf{x},\mathbf{y})\rangle$$

** Efficiently composable & computable from a circuit representation!

An equivalence checking methodology

Basic fact:

$$U = I \iff H^{\otimes n} U H^{\otimes n} |0\rangle = |0\rangle$$

To check equivalence of a circuit C w.r.t. a circuit or specification C',

1. Compute sum-over-paths representations U_C and $U_{C'}$
2. Construct quantum miter $U = H^{\otimes n} U_C \circ U_{C'}^\dagger H^{\otimes n}$
3. If

   $$U : |\mathbf{x}\rangle \mapsto \frac{1}{\sqrt{2}^k} \sum_{\mathbf{y}\in\{0,1\}^k} e^{2\pi i p(\mathbf{x},\mathbf{y})} |f(\mathbf{x},\mathbf{y})\rangle,$$

   verify

   $$\frac{1}{\sqrt{2}^k} \sum_{\mathbf{y}\in\{0,1\}^k, f(0,\mathbf{y})=0} e^{2\pi i p(0,\mathbf{y})} = 1$$

If $p \in \mathbb{Z}_2[\mathbf{x},\mathbf{y}]$, then step 3 reduces to #SAT. Moreover, if $\deg(p) \leq 2$, step 3 is efficiently computable (Montanaro, arXiv:1607.08473)

Symbolic reductions

Can we do better for other polynomials?

Recall: for Clifford+T, $p \in \mathbb{Z}_8[\mathbf{x}, \mathbf{y}]$

$$\frac{1}{\sqrt{2}^{k+1}} \sum_{\substack{y \in \{0,1\}^k \\ y' \in \{0,1\}}} \omega^{4y'q(x,y)+r(x,y)} |f(x,y)\rangle = \frac{1}{\sqrt{2}^{k-1}} \sum_{\substack{y \in \{0,1\}^k \\ q(x,y)=0}} \omega^{r(x,y)} |f(x,y)\rangle \quad (1)$$

$$\frac{1}{\sqrt{2}^{k+1}} \sum_{\substack{y \in \{0,1\}^k \\ y' \in \{0,1\}}} \omega^{2y'+4y'q(x,y)+r(x,y)} |f(x,y)\rangle = \frac{1}{\sqrt{2}^{k}} \sum_{y \in \{0,1\}^k} \omega^{1+6q(x,y)+r(x,y)} |f(x,y)\rangle \quad (2)$$

Using just relation (1), possible to verify a number of optimized arithmetic operators on 32-bit registers against specifications in seconds

Conclusion

- Formalized an irreversible language Revs
- Designed a new eager cleaning method based on cleanup expressions
- Implemented & formally verified a compiler (ReVerC) in F*

Take aways

- Proving theorems about real code is not unreasonably difficult
- Design code in such a way to minimize the scope of difficult logic

Going forward

Formally verify quantum circuit compilers

- Verifying library function implementations
- Verifying optimization

Develop methods for

- Functional coverage?

Thank you!

Questions?