

Verified compilation of space-efficient reversible circuits

Matthew Amy

*Institute for Quantum Computing and David R. Cheriton School of Computer Science,
University of Waterloo, Waterloo, ON, Canada*

Martin Roetteler and Krysta Svore

Microsoft Research, Redmond, WA, USA

The generation of reversible circuits from high-level code is an important problem in several application domains, including low-power electronics and quantum computing. Existing tools compile and optimize reversible circuits for various metrics, such as the overall circuit size or the total amount of space required to implement a given function reversibly. However, little effort has been spent on verifying the correctness of the results, an issue of particular importance in quantum computing. There, compilation allows not only mapping to hardware, but also the estimation of resources required to implement a given quantum algorithm. This resource determination is crucial for identifying which algorithms will outperform their classical counterparts. We present a reversible circuit compiler called REVER, which has been formally verified in F* and compiles circuits that operate correctly with respect to the input program. Our compiler compiles the REVS language [1] to combinational reversible circuits with as few *ancillary* bits as possible, and provably cleans temporary values.

I. INTRODUCTION

The ability to evaluate classical functions coherently and in superposition as part of a larger quantum computation is essential for many quantum algorithms. For example, Shor’s quantum algorithm [2] uses classical modular arithmetic and Grover’s quantum algorithm [3] uses classical predicates to implicitly define the underlying search problem. There is a resulting need for tools to help a programmer translate classical, irreversible programs into a form which a quantum computer can understand and carry out, namely into reversible circuits which are a special case of quantum transformations [4]. Other applications of reversible computing include low-power design of classical circuits, see [5] for background and a critical discussion.

Several tools have been developed for synthesizing reversible circuits, ranging from low-level methods for small circuits such as [6–10] (see also [11] for a survey) to high-level programming languages and compilers [1, 12–16]. In this paper we are interested in the latter class—i.e., methods for compiling high-level code to reversible circuits. Such compilers commonly perform optimization, as the number of bits quickly grows with the standard techniques for achieving reversibility (see, e.g., [17]). The crucial question, as with general purpose compilers, is whether or not we can trust these optimizations.

In most cases, extensive testing of compiled programs is sufficient to establish the correctness of both the source program and its translation to a target architecture by the compiler. Formal methods are typically reserved for safety- (or mission-) critical applications. For instance, formal verification is an essential step in modern computer-aided circuit design due largely to the high cost of a recall. Reversible – specifically, quantum – circuits occupy a different design space in that 1) they are typically “software circuits,” i.e., they are not intended to be implemented directly in hardware, and 2) there exists few examples of hardware to actually run such circuits. Given that there are no large-scale universal quantum computers currently in existence, one of the goals of writing a quantum circuit compiler at all is to accurately gauge the amount of physical resources needed to perform a given algorithm, a process called *resource estimation*. Such resource estimates can be used to identify the “crossover point” when a problem becomes more efficient to solve on a quantum computer, and are invaluable both in guiding the development of quantum computers and in assessing their potential impact. However, different compilers give wildly different resource estimates for the same algorithms, making it difficult to trust that the reported numbers are correct. For this reason compiled circuits need to have some level of formal guarantees as to their correctness for resource estimation to be effective.

In this paper we present REVER, a lightly optimizing compiler for the REVS language [1] which has been written and proven correct in the dependently typed language F* [18]. Circuits compiled with REVER are

verified to preserve the semantics of the source REVS program, which we have for the first time formalized, and to reset or *clean* all ancillary (temporary) bits used so that they may be used later in other computations. In addition to formal verification of the compiler, REVER provides an assertion checker which can be used to formally verify the source program itself, allowing effective end-to-end verification of reversible circuits. We also develop a type inference algorithm for REVS mixing subtyping and record concatenation, a known hard problem [19]. Following is a summary of the contributions of our paper:

- We create a formal semantic model of REVS which respects the original compiler.
- We present a compiler for REVS called REVER, written in F*. The compiler currently has two modes: direct to circuit and Boolean expression compilation, the latter of which is space efficient.
- We provide a type inference algorithm for REVS, used to infer and allocate circuit inputs. Our algorithm generates and then solves a set of type constraints using *bound maps* [20].
- Finally, we verify correctness of REVER with machine-checked proofs that the compiled reversible circuits faithfully implement the input program’s semantics, and that all ancillas used are returned to their initial state.

a. Related work Due to the reversibility requirement of quantum computing, quantum programming languages and compilers typically have a methods for generating reversible circuits. Quantum programming languages typically allow compilation of classical, irreversible code in order to minimize the effort of porting existing code into the quantum domain. In QCL [21], “pseudo-classical” operators – classical functions meant to be run on a quantum computer – are written in an imperative style and compiled with automatic ancilla management. As in REVS, such code manipulates registers of bits, splitting off sub-registers and concatenating them together. The more recent Quipper [15] automatically generates reversible circuits from classical code by a process called *lifting*: using Haskell metaprogramming, Quipper lifts the classical code to the reversible domain with automated ancilla management. However, little space optimization is performed [17].

Other approaches to high-level synthesis of reversible circuits are based on writing code in *reversible programming languages* – that is, the programs themselves are written in a reversible way. Perhaps most notable in this line of research is Janus [12] and its various extensions [13, 14, 16]. These languages typically feature a *reversible update* and some bi-directional control operators, such as if statements with exit assertions. Due to the presence of dynamic control in the source language, such languages typical target reversible instruction set architectures like Pendulum [22], as opposed to the combinational circuits that REVER and in general, quantum compilers target.

Verification of reversible circuits has been previously considered from the viewpoint of checking equivalence against a benchmark circuit or specification [23, 24]. This can double as both *program verification* and *translation validation*, but every compiled circuit needs to be verified separately. Moreover, a program that is easy to formally verify may be translated into a circuit with hundreds of bits, and is thus very difficult to verify. To the authors’ knowledge, no verification of a reversible circuit compiler has yet been carried out. By contrast, many compilers for general purpose programming languages have been formally verified in recent years – most famously, the CompCert optimizing C compiler [25, 26], written and verified in Coq. Since then, many other compilers have been developed and verified in a range of languages and logics including Coq, HOL, F*, etc., with features such as shared memory [27], functional programming [28–30] and modularity [31, 32].

b. Outline The rest of our paper is organized as follows: in the following section we provide an overview of reversible computing and the original REVS compiler. Section III formalizes REVS and introduces a semantic model, as well as our internal and target languages. Section IV describes our compilation methods and Section V details our type system and inference algorithm. Section VI gives our machine-checked correctness proofs for REVER. Finally, Section VII compares our compiled circuits to those of REVS, and Section VIII concludes the paper.

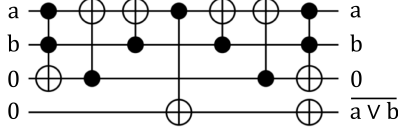


FIG. 1. A Toffoli network computing the NOR function $f(a, b) = \overline{a \vee b}$. When read from left to right and numbering the qubits from top to bottom as 0, 1, 2, 3, the applied gates are Toffoli 0 1 2, CNOT 2 0, CNOT 1 0, CNOT 0 3, CNOT 1 0, CNOT 2 0, Toffoli 0 1 2, and NOT 3. Bit 2 is initialized in the zero state and returned clean, i.e., in the zero state whereas bit 3 is used to store the result $f(a, b)$.

II. REVERSIBLE COMPUTING

Reversible functions are Boolean functions $f : \{0, 1\}^n \rightarrow \{0, 1\}^n$ which can be inverted on all outputs, i.e., precisely those functions which correspond to permutations of a set of cardinality 2^n , for some $n \in \mathbb{N}$. As with classical circuits, reversible functions can be constructed from universal gate sets – for instance, it is known that the Toffoli gate which maps $(x, y, z) \mapsto (x, y, z \oplus (x \wedge y))$, together with the controlled-NOT gate (CNOT) which maps $(x, y) \mapsto (x, x \oplus y)$ and the NOT gate which maps $x \mapsto x \oplus 1$, is universal for reversible computation [4]. In particular, any given target function, when considered as a permutation π_f can be implemented over this gate set at the expense of at most 1 additional bit.

The number of Toffoli gates used in the implementation of a given permutation π_f is the basic measure of complexity. Indeed, it is known from the theory of fault-tolerant quantum computing [4] that the Toffoli gate has a substantial cost, whereas the cost of so-called Clifford gates, such as CNOT and NOT, can usually be neglected. Another important metric that is associated to a reversible circuit is the amount of scratch space required to implement a given target function, i.e., temporary bits which store intermediate results of a computation. In quantum computing such bits are commonly denoted as *ancilla* bits. A very important difference to classical computing is that scratch bits cannot just be overwritten when they are no longer needed: any ancilla that is used as scratch space during a reversible computation must be returned to its initial value—commonly assumed to be 0—computationally. Moreover, if an ancilla bit is not “cleaned” in this way, in a quantum computation it may remain entangled with the computational registers which in turn can destroy the desired interferences that are crucial for many quantum algorithms.

Figure 1 shows a reversible circuit over NOT, CNOT, and Toffoli gates computing the NOR function. NOT gates are denoted by an \oplus , while CNOT and Toffoli gates are written with an \oplus on the *target* bit (the bit whose value changes) connected by a vertical line to, respectively, either one or two *control* bits identified by solid dots. Two ancilla qubits are used which both initially are 0; one of these ultimately holds the (otherwise irreversible) function value, the other is returned to zero. For larger circuits, it becomes a non-trivial problem to assert a) that indeed the correct target function f is implemented and b) that indeed all ancillas that are not outputs are returned to 0.

c. **REVS** From Bennett’s work on reversible Turing machines follows that any function can be implemented by a suitable reversible circuit [33]: if an n -bit function $x \mapsto f(x)$ can be implemented with K gates over $\{\text{NOT}, \text{AND}\}$, then the reversible function $(x, y) \mapsto (x, y \oplus f(x))$ can be implemented with at most $2K + n$ gates over the Toffoli gate set. The basic idea behind Bennett’s method is to replace all AND gates with Toffoli gates, then perform the computation, copy out the result, and undo the computation. This strategy is illustrated in Figure 2, where the box labelled U_f corresponds to f with all AND gates substituted with Toffoli gates and the inverse box is simply obtained by reversing the order of all gates in U_f . Bennett’s method has been implemented in the Haskell-based quantum programming language Quipper [15, 34] via a monad that allows lifting of classical functions to Toffoli networks with the Bennett method. One potential disadvantage of Bennett’s method is the large number of ancillas it requires as the required memory scales proportional to the circuit *size* of the initial, irreversible function f . In a recent work, an attempt was made with the REVS compiler (and programming language of the same name) [1] to improve on the space-complexity of Bennett’s strategy by generating circuits that are *space-efficient* – that is, REVS is an optimizing compiler with respect to the number of bits used. We build on their work in this paper,

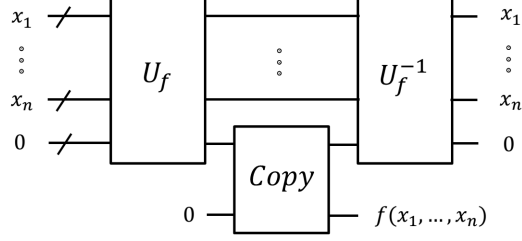


FIG. 2. A reversible circuit computing $f(x_1, \dots, x_m)$ using the Bennett trick.

Var x
Nat $i, j \in \mathbb{N}$
Loc $l \in \mathbb{N}$
Val $v ::= \text{unit} \mid l \mid \text{register } l_1 \dots l_n \mid \lambda x. t$
Term $t ::= \text{let } x = t_1 \text{ in } t_2 \mid \lambda x. t \mid (t_1 t_2) \mid t_1; t_2 \mid x \mid t_1 \leftarrow t_2$
 $\mid \text{true} \mid \text{false} \mid t_1 \oplus t_2 \mid t_1 \wedge t_2 \mid \text{clean } t \mid \text{assert } t$
 $\mid \text{register } t_1 \dots t_n \mid t.[i] \mid t.[i..j] \mid \text{append } t_1 t_2 \mid \text{rotate } i t$

FIG. 3. Syntax of REVS.

formalizing REVS and developing a *verified* compiler without too much loss in efficiency.

III. LANGUAGES

In this section we give a formal definition of REVS, as well as the intermediate and target languages of the compiler.

A. The Source

The abstract syntax of REVS is presented in Figure 3. The core of the language is a simple imperative language over Boolean and array (register) types. The language is further extended with ML-style functional features, namely first-class functions and *let* definitions, and a reversible domain-specific construct *clean*.

In addition to the basic syntax of Figure 3 we add the following derived operations

$$\begin{aligned}
 \neg t &\triangleq \text{true} \oplus t, & t_1 \vee t_2 &\triangleq (t_1 \wedge t_2) \oplus (t_1 \oplus t_2), \\
 \text{if } t_1 \text{ then } t_2 \text{ else } t_3 &\triangleq (t_1 \wedge t_2) \oplus (\neg t_1 \wedge t_3), \\
 \text{for } x \text{ in } i..j \text{ do } t &\triangleq t[x \mapsto i]; \dots; t[x \mapsto j].
 \end{aligned}$$

Note that REVS has no *dynamic* control. This is due to the restrictions of our target architecture (see below).

The REVER compiler uses F# as a meta-language to generate REVS code with particular register sizes and indices, possibly computed by some more complex program. Writing an F# program that generates REVS code is similar in effect to writing in a hardware description language [35]. We use F#'s *quotations* mechanism to achieve this by writing REVS programs in quotations $\langle @ \dots @ \rangle$. Note that unlike languages

```

1  let s0 a =
2    let a2 = rot 2 a
3    let a13 = rot 13 a
4    let a22 = rot 22 a
5    let t = Array.zeroCreate 32
6    for i in 0 .. 31 do
7      t.[i] <- a2.[i] <> a13.[i] <> a22.[i]
8    t
9  let s1 a =
10   let a6 = rot 6 a
11   let a11 = rot 11 a
12   let a25 = rot 25 a
13   let mutable t = Array.zeroCreate 32
14   for i in 0 .. 31 do
15     t.[i] <- a6.[i] <> a11.[i] <> a25.[i]
16   t
17  let ma a b c =
18   let t = Array.zeroCreate 32
19   for i in 0 .. 31 do
20     t.[i] <- (a.[i] && (b.[i] <> c.[i]))
21             <> (b.[i] && c.[i])
22   t
23  let ch e f g =
24   let t = Array.zeroCreate 32
25   for i in 0 .. 31 do
26     t.[i] <- e.[i] && f.[i] && g.[i]
27   t
29  fun k w x ->
30   let hash x =
31     let a = x.[0..31], b = x.[32..63]
32     c = x.[64..95], d = x.[96..127],
33     e = x.[128..159], f = x.[160..191],
34     g = x.[192..223], h = x.[224..255]
35     (%modAdd 32) (ch e f g) h
36     (%modAdd 32) (s0 a) h
37     (%modAdd 32) w h
38     (%modAdd 32) k h
39     (%modAdd 32) h d
40     (%modAdd 32) (ma a b c) h
41     (%modAdd 32) (s1 e) h
42   for i in 0 .. n - 1 do
43     hash (rot 32*i x)
44   x

```

FIG. 4. Implementation of the SHA-2 algorithm with n rounds, using a meta-function `modAdd`.

such as Quipper, our strictly combinational target architecture doesn't allow computations in the meta-language to depend on computations within REVS.

Example 1. The 256-bit Secure Hash Algorithm 2 [36] is a cryptographic hash algorithm which performs a series of modular additions and functions on 32-bit chunks of a 256-bit hash value, before rotating the register and repeating for a number of rounds. In the REVS implementation, shown in Figure 4, this is achieved by a function `hash` which takes a length 256 register, then makes calls to modular adders with different 32-bit slices. At the end of the round, the entire register is rotated 32 bits with the `rotate` command.

d. Semantics We designed our semantic model of REVS with two goals in mind:

1. keep the semantics as close to the original implementation, REVS, as possible, and
2. simplify the task of formal verification.

The result is a somewhat non-standard semantics that is nonetheless intuitive for the programmer. Moreover, the particular semantic model naturally enforces a style of programming that results in efficient circuits and allows common design patterns to be optimized.

The operational semantics of REVS is presented in big-step style in Figure 5 as a relation $\Rightarrow_C \text{Config} \times \text{Config}$ on configurations – pairs of terms and Boolean-valued stores. A key feature of our semantic model is that Boolean, or bit values are always allocated on the store. Specifically, Boolean constants and expressions are modelled by allocating a new location on the store to hold its value – as a result all Boolean values, including constants, are mutable. All other values are immutable.

The allocation of Boolean values on the store serves two main purposes: to give the programmer fine-grain control over how many bits are allocated, and to provide a simple and efficient model of *registers* – arrays of bits. In studying pseudo-code for low-level bitwise operations, e.g., arithmetic and cryptographic functions, we observed that most algorithms were naturally specified by a combination of array-type accesses – modifying individual elements of bit arrays – and list-type manipulations – splitting and merging bit arrays. While these patterns could be implemented with arrays, most algorithms would require either complex index book-keeping by the programmer or extraneous copying of arrays; in the latter case, circuits could be optimized to do away with unnecessary copies, but this complicates the process of formal verification. In particular, we want to compile efficient circuits with as little extrinsic optimization as possible.

$$\begin{array}{c}
\text{Store } \sigma : \mathbb{N} \rightarrow \mathbb{B} \\
\text{Config } c ::= \langle t, \sigma \rangle
\end{array}$$

$$\begin{array}{c}
[\text{REFL}] \frac{}{\langle v, \sigma \rangle \Rightarrow \langle v, \sigma \rangle} \quad [\text{LET}] \frac{\langle t_1, \sigma \rangle \Rightarrow \langle v_1, \sigma' \rangle \quad \langle t_2[x \mapsto v_1], \sigma' \rangle \Rightarrow \langle v_2, \sigma'' \rangle}{\langle \text{let } x = t_1 \text{ in } t_2, \sigma \rangle \Rightarrow \langle v_2, \sigma'' \rangle} \\
[\text{APP}] \frac{\langle t_1, \sigma \rangle \Rightarrow \langle \lambda x. t'_1, \sigma' \rangle \quad \langle t_2, \sigma' \rangle \Rightarrow \langle v_2, \sigma'' \rangle \quad \langle t'_1[x \mapsto v_2], \sigma'' \rangle \Rightarrow \langle v, \sigma''' \rangle}{\langle (t_1 \ t_2), \sigma \rangle \Rightarrow \langle v, \sigma''' \rangle} \quad [\text{SEQ}] \frac{\langle t_1, \sigma \rangle \Rightarrow \langle \text{unit}, \sigma' \rangle \quad \langle t_2, \sigma' \rangle \Rightarrow \langle v, \sigma'' \rangle}{\langle t_1; t_2, \sigma \rangle \Rightarrow \langle v, \sigma'' \rangle} \\
[\text{ASSN}] \frac{\langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma' \rangle \quad \langle t_2, \sigma' \rangle \Rightarrow \langle l_2, \sigma'' \rangle}{\langle t_1 \leftarrow t_2, \sigma \rangle \Rightarrow \langle \text{unit}, \sigma''[l_1 \mapsto \sigma''(l_2)] \rangle} \quad [\text{BEXP}] \frac{\langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma' \rangle \quad \langle t_2, \sigma' \rangle \Rightarrow \langle l_2, \sigma'' \rangle \quad l_3 \notin \text{dom}(\sigma'')}{\langle t_1 \star t_2, \sigma \rangle \Rightarrow \langle l_3, \sigma''[l_3 \mapsto \sigma''(l_1) \star \sigma''(l_2)] \rangle} \\
[\text{TRUE}] \frac{l \notin \text{dom}(\sigma)}{\langle \text{true}, \sigma \rangle \Rightarrow \langle l, \sigma''[l \mapsto 1] \rangle} \quad [\text{FALSE}] \frac{l \notin \text{dom}(\sigma)}{\langle \text{false}, \sigma \rangle \Rightarrow \langle l, \sigma''[l \mapsto 0] \rangle} \\
[\text{APPEND}] \frac{\langle t_1, \sigma \rangle \Rightarrow \langle \text{register } l_1 \dots l_m, \sigma' \rangle \quad \langle t_2, \sigma' \rangle \Rightarrow \langle \text{register } l_{m+1} \dots l_n, \sigma'' \rangle}{\langle \text{append } t_1 \ t_2, \sigma \rangle \Rightarrow \langle \text{register } l_1 \dots l_n, \sigma'' \rangle} \\
[\text{INDEX}] \frac{\langle t, \sigma \rangle \Rightarrow \langle \text{register } l_1 \dots l_n, \sigma' \rangle \quad 1 \leq i \leq n}{\langle t.[i], \sigma \rangle \Rightarrow \langle l_i, \sigma' \rangle} \quad \begin{array}{c} \langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma_1 \rangle \\ \langle t_2, \sigma \rangle \Rightarrow \langle l_2, \sigma_2 \rangle \\ \vdots \\ \langle t_n, \sigma \rangle \Rightarrow \langle l_n, \sigma_n \rangle \end{array} \\
[\text{SLICE}] \frac{\langle t, \sigma \rangle \Rightarrow \langle \text{register } l_1 \dots l_n, \sigma' \rangle \quad 1 \leq i \leq j \leq n}{\langle t.[i..j], \sigma \rangle \Rightarrow \langle \text{register } l_i \dots l_j, \sigma' \rangle} \quad [\text{REG}] \frac{\langle \text{register } t_1 \dots t_n, \sigma \rangle \Rightarrow \langle \text{register } l_1 \dots l_n, \sigma_n \rangle}{\langle \text{register } t_1 \dots t_n, \sigma \rangle \Rightarrow \langle \text{register } l_1 \dots l_n, \sigma_n \rangle} \\
[\text{ROTATE}] \frac{\langle t, \sigma \rangle \Rightarrow \langle \text{register } l_1 \dots l_n, \sigma' \rangle \quad 1 < i < n}{\langle \text{rotate } t \ i, \sigma \rangle \Rightarrow \langle \text{register } l_i \dots l_{i-1}, \sigma' \rangle} \\
[\text{CLEAN}] \frac{\langle t, \sigma \rangle \Rightarrow \langle l, \sigma' \rangle \quad \sigma'(l) = \text{false}}{\langle \text{clean } t, \sigma \rangle \Rightarrow \langle \text{unit}, \sigma'_{\text{dom}(\sigma') \setminus \{l\}} \rangle} \quad [\text{ASSERT}] \frac{\langle t, \sigma \rangle \Rightarrow \langle l, \sigma' \rangle \quad \sigma'(l) = \text{true}}{\langle \text{assert } t, \sigma \rangle \Rightarrow \langle \text{unit}, \sigma' \rangle}
\end{array}$$

FIG. 5. Operational semantics of REVS.

The semantics of \oplus and \wedge are also notable in that they first reduce both arguments to locations, *then* retrieve their value. This allows programmers to write statements whose value may not be immediately apparent – e.g., $x \oplus (x \leftarrow y; y)$, which under these semantics will always evaluate to 0. The benefit of this definition is that it allows the compiler to perform important optimizations without a significant burden on the programmer (see Section IV).

Semantically, the `clean` and `assert` commands have no effect, other than to evaluate their argument. In the case of the `assert` statement, while an assertion could be modelled with reversible circuits, it is unlikely that any circuit design would ever have a use for such a construct; instead, assertions are used strictly for program verification by the compiler. `Clean` statements have a similar interpretation – rather than having the compiler generate circuit code cleaning a particular bit, they are viewed as compiler directives asserting that a particular bit is in the zero state. The location is then freed for use by the rest of the program, making the statement analogous to `free` in other languages. Practically speaking this statement is only useful in the form `clean x` for some identifier x .

B. Boolean expressions

Our compiler uses XOR-AND Boolean expressions – single output classical circuits over XOR and AND gates – as an intermediate language. Compilation from Boolean expressions into reversible circuits forms the main “code generation” step of our compiler. We briefly describe its syntax and semantics.

A Boolean expression is defined as an expression over Boolean constants, variable indices, and logical \oplus

and \wedge operators. Explicitly, we define

$$\mathbf{BExp} \ B ::= 0 \mid 1 \mid i \in \mathbb{N} \mid B_1 \oplus B_2 \mid B_1 \wedge B_2.$$

Note that we use the symbols $0, 1, \oplus$ and \wedge interchangeably with their interpretation in the Boolean field \mathbb{F}_2 . We use $\text{vars}(B)$ to refer to the set of variables used in B .

We interpret a Boolean expression as a function from (total) Boolean-valued states to Booleans. In particular, we define $\mathbf{State} = \mathbb{N} \rightarrow \mathbb{B}$ and denote the semantics of a Boolean expression by $\llbracket B \rrbracket : \mathbf{State} \rightarrow \mathbb{B}$. The formal definition of $\llbracket B \rrbracket$ is obvious, but we include it below for completeness:

$$\begin{aligned} \llbracket 0 \rrbracket s &= 0 & \llbracket 1 \rrbracket s &= 1 & \llbracket i \rrbracket s &= s(i) \\ \llbracket B_1 \oplus B_2 \rrbracket s &= \llbracket B_1 \rrbracket s \oplus \llbracket B_2 \rrbracket s \\ \llbracket B_1 \wedge B_2 \rrbracket s &= \llbracket B_1 \rrbracket s \wedge \llbracket B_2 \rrbracket s \end{aligned}$$

C. Target architecture

REVER compiles to *combinational, reversible circuits* over NOT, controlled-NOT and Toffoli gates. By combinational circuits we mean a series of logic gates applied to bits with no external means of control or memory – effectively pure logical functions. We chose this model as it is suitable for implementing classical functions and oracles within quantum computations [4].

Formally, we define

$$\mathbf{Circ} \ C ::= - \mid \text{NOT } i \mid \text{CNOT } i \ j \mid \text{Toffoli } i \ j \ k \mid C_1 :: C_2,$$

i.e., \mathbf{Circ} is the free monoid over NOT, CNOT, and Toffoli gates with unit $-$ and the append operator $::$. All but the last bit in each gate is called a *control*, whereas the final bit is denoted as the *target*. Only the target is modified by a gate. We use $\text{use}(C)$, $\text{mod}(C)$ and $\text{control}(C)$ to denote the set of bit indices that are used in, modified by, or used as a control in the circuit C , respectively. A circuit is *well-formed* if no gate contains more than one reference to a bit – i.e., the bits used in each Controlled-Not or Toffoli gate are distinct.

Similar to Boolean expressions, a circuit is interpreted as a function from states (maps from indices to Boolean values) to states, given by applying each gate which updates the previous state in order. The formal definition of the semantics of a reversible circuit C , given by $\llbracket C \rrbracket : \mathbf{State} \rightarrow \mathbf{State}$, is straightforward:

$$\begin{aligned} \llbracket \text{NOT } i \rrbracket s &= s[i \mapsto \neg s(i)] \\ \llbracket \text{CNOT } i \ j \rrbracket s &= s[j \mapsto s(i) \oplus s(j)] \\ \llbracket \text{Toffoli } i \ j \ k \rrbracket s &= s[k \mapsto (s(i) \wedge s(j)) \oplus s(k)] \\ \llbracket - \rrbracket s &= s & \llbracket C_1 :: C_2 \rrbracket s &= (\llbracket C_2 \rrbracket \circ \llbracket C_1 \rrbracket) s \end{aligned}$$

We use $s[x \mapsto y]$ to denote that function that maps x to y , and all other inputs z to $s(z)$; by an abuse of notation we use $[x \mapsto y]$ to denote other substitutions as well.

IV. COMPILATION

In this section we discuss the implementation of REVER. The compiler consists of 4381 lines of non-blank code in a common subset of F* and F#, with a front-end to evaluate and translate F# quotations into REVS expressions. The core of the compiler – i.e., compilation from the abstract syntax of REVS into reversible circuits – was proven correct with F*.

A. Boolean expression compilation

The core of REVER’s code generation is a compiler from Boolean expressions into reversible circuits. We effectively use the method employed in REVS [1], with the addition of an eager cleanup option.

As a Boolean expression is already in the form of an irreversible classical circuit, the main job of the compiler is to allocate ancillas to store sub-expressions whenever necessary. REVER does this by maintaining a global *ancilla heap* ξ which keeps track of the currently available (zero-valued) ancillary bits. Cleaned ancillas (ancillas returned to the zero state) may be pushed back onto the heap, and allocations have been formally verified to return previously used ancillas if any are available, hence not using any extra space.

The function `compile-BExp`, shown in pseudo-code in Algorithm 1, takes a Boolean expression B and a target bit i then generates a reversible circuit computing $i \oplus B$. Note that ancillas are only allocated to store sub-expressions of \wedge expressions, since \oplus is associative (i.e., $i \oplus (B_1 \oplus B_2) = (i \oplus B_1) \oplus B_2$).

Algorithm 1 Boolean expression compiler.

```

function COMPILER-BEXP( $B, i$ ) // Global ancilla heap  $\xi$ 
  if  $B = 0$  then –
  else if  $B = 1$  then NOT  $i$ 
  else if  $B = j$  then CNOT  $j i$ 
  else if  $B = B_1 \oplus B_2$  then
    compile-BExp( $B_1, i$ ) :: compile-BExp( $B_2, i$ )
  else //  $B = B_1 \wedge B_2$ 
     $a_1 \leftarrow \text{pop}(\xi), C_1 \leftarrow \text{compile-BExp}(B_1, a_1)$ 
     $a_2 \leftarrow \text{pop}(\xi), C_2 \leftarrow \text{compile-BExp}(B_2, a_2)$ 
     $C_1 :: C_2 :: \text{Toffoli } a_1 a_2 i$ 
  end if
end function

```

e. Cleanup The definition of `COMPILER-BEXP` above is adequate to generate a reversible circuit computing the Boolean expression, but it leaves many garbage bits that take up space and need to be cleaned before they can be re-used. REVS provides two main modes of cleanup for Boolean expressions: eager and lazy. The former performs cleanup after each recursive call, while the latter performs one cleanup after the entire Boolean expression is computed. The key difference is that the eager strategy uses fewer bits, but requires more gates as sub-expressions may need to be re-computed. While we only implement these two strategies, the problem of finding a good tradeoff between bits used and number of gates can be formalized as *pebble games* [37].

To facilitate the cleanup – or *uncomputing* – of a circuit, we define the *restricted inverse uncompute*(C, A) of C with respect to a set of bits $A \subset \mathbb{N}$ by reversing the gates of C , and removing any gates with a target in A . For instance:

$$\text{uncompute}(\text{CNOT } i j, A) = \begin{cases} - & \text{if } j \in A \\ \text{CNOT } i j & \text{otherwise} \end{cases}$$

The other cases are defined similarly.

The restricted inverse allows the temporary values of a reversible computation to be uncomputed without affecting any of the target bits. In particular, if $C = \text{compile-BExp}(B, i)$, then the circuit $C :: \text{uncompute}(C, \{i\})$ maps a state s to $s[i \mapsto \llbracket B \rrbracket s \oplus s(i)]$. Intuitively, since no bits contained in A are modified, the restricted inverse preserves their values; that the restricted inverse uncomputes the values of the remaining bits is more difficult to see, but it can be observed that if the computation doesn’t *depend* on the value of a bit in A , the computation will be inverted. We formalize and prove this statement in Section VI. The eager and lazy cleanup strategies are defined respectively by applying this cleanup at every recursive call to `compile-BExp` or to just the outer-most call. In order to re-use ancilla bits that have been cleaned, they are pushed back onto the heap.

B. REVS compilation

In studying the REVS compiler, we observed that most of what the compiler was doing was evaluating the non-Boolean parts of the program – effectively bookkeeping for registers – only generating circuits for a small kernel of cases. As a result, transformations to different Boolean representations (e.g., circuits, dependence graphs [1]) as well as the interpreter itself reused significant portions of this bookkeeping code. To make use of this redundancy to simplify both writing and verifying the compiler, we designed REVER as a *partial evaluator* parameterized by an abstract machine. As a side effect, we arrive at a unique and intuitive model for circuit compilation similar to staged computation (see, e.g., [38]).

Our compiler works by evaluating the program with an abstract machine providing mechanisms for initializing and assigning locations on the store to Boolean expressions. In particular, an *interpretation* \mathcal{I} consists of a domain D and 2 operations:

$$\begin{aligned} \text{assign} &: D \times \mathbb{N} \times \mathbf{BExp} \rightarrow D \\ \text{eval} &: D \times \mathbb{N} \times \mathbf{State} \rightarrow \mathbb{B}. \end{aligned}$$

Formally, a series of assignments in an interpretation builds a Boolean computation or circuit within a specific model (i.e., classical, reversible, different gate sets) which may be simulated on an initial state with the `eval` function – effectively an operational semantics of the model. Practically speaking, it abstracts the store in Figure 5 and allows delayed computation or additional processing of the Boolean expression stored in a cell, which may be mapped into reversible circuits immediately or after the entire program has been evaluated. We give some examples of interpretations below.

Example 2. The standard interpretation $\mathcal{I}_{\text{standard}}$ has domain $\mathbf{Store} = \mathbb{N} \rightarrow \mathbb{B}$, together with the operations

$$\begin{aligned} \text{assign}_{\text{standard}}(\sigma, l, B) &= \sigma[l \mapsto \llbracket B \rrbracket \sigma] \\ \text{eval}_{\text{standard}}(\sigma, l, s) &= \sigma(l). \end{aligned}$$

Partial evaluation over the standard interpretation coincides exactly with the operational semantics of REVS.

Example 3. The *Boolean expression* interpretation $\mathcal{I}_{\text{BExp}}$ has domain $\mathbb{N} \rightarrow \mathbf{BExp}$, together with the operations

$$\begin{aligned} \text{assign}_{\text{BExp}}(\sigma, l, B) &= \sigma[l \mapsto B[l' \in B \mapsto \sigma(l')]] \\ \text{eval}_{\text{BExp}}(\sigma, l, s) &= \llbracket \sigma(l) \rrbracket s. \end{aligned}$$

The Boolean expression interpretation can be viewed as the standard interpretation with deferred evaluation – that is, it expands Boolean expressions over *locations* to Boolean expressions over a set of variables representing the program inputs, but defers evaluation until a set of initial values are supplied. Effectively the result is the expression (or expressions) the program computes on a given input.

Example 4. The *reversible circuit* interpretation $\mathcal{I}_{\text{circuit}}$ is somewhat more complex. It has domain $(\mathbb{N} \rightarrow \mathbb{N}) \times \mathbf{Circ}$, where the first element maps program-level locations to circuit-level bits and the second is a reversible circuit. The 2 operations are defined as follows – note that `assign` uses the global ancilla heap ξ :

$$\begin{aligned} \text{assign}_{\text{circuit}}((\rho, C), l, B) &= (\rho[l \mapsto i], C \ :: \ C') \\ \text{where } i &= \text{pop}(\xi), \\ C' &= \text{compile-BExp}(B[l' \in \text{vars}(B) \mapsto \rho(l')], i) \\ \text{eval}_{\text{circuit}}((\rho, C), l, s) &= s'(\rho(l)), \quad \text{where } s' = \llbracket C \rrbracket s. \end{aligned}$$

This interpretation generates a reversible circuit directly from the program by maintaining a mapping from program locations to bits. When a location is overwritten, a new ancilla i is allocated and the expression $B \oplus i$ is compiled into a circuit. Evaluation amounts to running the circuit on an initial state, then retrieving the value at a particular bit.

REVER provides implementations of the interpretations above. It is also possible to define an interpretation that generates *mutable dependence diagrams*, which were used as an intermediate representation in [1] to compile space-efficient reversible circuits.

Given an interpretation \mathcal{I} with domain D , we define the set of \mathcal{I} -configurations as $\mathbf{Config}_{\mathcal{I}} = \mathbf{Term} \times D$ – that is, \mathcal{I} -configurations are pairs of programs and elements of D which function as the heap. The relation

$$\Rightarrow_{\mathcal{I}} \subseteq \mathbf{Config}_{\mathcal{I}} \times \mathbf{Config}_{\mathcal{I}}$$

gives the operational semantics of REVS over the interpretation \mathcal{I} . We do not give a formal definition of $\Rightarrow_{\mathcal{I}}$, as it can be obtained trivially from the definition of \Rightarrow (Figure 5) by replacing all heap updates with `assign`. To compile a program term t , REVER evaluates t over a particular interpretation \mathcal{I} (for instance, the reversible circuit interpretation) and a suitable initial heap $\sigma \in D$ according to the semantic relation $\Rightarrow_{\mathcal{I}}$. In this way, evaluating a program and compiling a program to, for instance, a circuit look almost identical. As we will show later, this greatly simplifies the problem of verification.

Our compiler currently supports two modes of compilation: a default mode, and a space-efficient mode. The default mode evaluates the program using the circuit interpretation, and simply returns the circuit and output bit(s). The space-efficient mode, on the other hand, evaluates the program to a list of Boolean expressions over the inputs, by partial evaluation with the Boolean expression interpretation. The Boolean expression compiler is then used to compile each Boolean expression into a circuit. This method effectively erases the program structure, allowing the Boolean expression compiler to more effectively manage the allocation of ancillas for sub-expressions and to apply optimizations, described below. However, the duplication of sub-expressions leads to an exponential blowup, making this compilation method not scalable.

f. Input vs parameters Quipper [15] makes a distinction between function *inputs* and *parameters*, the latter of which are available at compile time. In this way the language provides a static, structural divide between run-time and compile-time computations. Partial evaluation on the other hand effectively erases this distinction, any structural limitations owing to the limitations of the target architectures (for instance, unbounded for loops cannot be compiled to a combinational circuit). Practically speaking, this means the programmer can use the same code to perform more or less computation at the circuit level – for instance, an adder may be partially instantiated to compile a circuit performing addition by a constant, without making any changes to the code itself. The downside of this method is it can be more difficult for the programmer to determine what the compiled circuit is actually computing. We intend to add more support for types and constructs which may be computed either at compile-time or run-time, such as integers and array indexing.

C. Optimization passes

Both methods of compilation described above suffer from the problem that more ancillas may be allocated than are actually needed, due to the program structure. Our implementation performs some optimizations, all of which have been verified to be correct, in order to reduce ancilla usage for some common patterns.

g. XOR-equal In the circuit interpretation, a new ancilla is allocated by default whenever a location is updated, since updates are generally not reversible. However, in the case when an update can be rewritten as an XOR of the target bit, the update can be performed reversibly. Specifically, given a Boolean expression B and target bit i , we try to factor B as $i \oplus B'$ where $i \notin \text{vars}(B')$, a condition required for the correctness of compile-BExp (see Section VI). If such a factoring exists, rather than allocate an ancilla to store the result we compile the expression B' with target i . In practice this significantly reduces the number of ancillas used, as evidenced by the REVS code in Figure 4 which makes extensive use of the XOR-equal pattern.

h. Boolean simplifications Before compiling a Boolean expression, we perform basic simplifications, e.g., short circuiting AND's and XOR's whenever possible. We also distribute ANDs to rewrite the expression in positive-polarity *ESOP* (*exclusive sum-of-products*) form. The benefit of this transformation is that it produces circuits with the minimal number of ancillas using compile-BExp without eager cleanup – in particular, the compiled circuit uses $k - 2$ ancilla bits where k is the highest degree of any product in the ESOP form. Its minimality with respect to compile-BExp can be observed by noting that the positive-polarity ESOP form of a Boolean function is unique, so any equivalent expression over AND and XOR must contain a product of length k and hence use at least $k - 2$ ancilla bits.

If cleanup is performed eagerly the number of bits used depends on the factoring of a given product into binary products. In particular, the product $((((a \wedge b) \wedge c) \wedge d) \wedge e) \wedge f$ would require 4 ancilla bits using the eager cleanup scheme, while the product $((a \wedge b) \wedge (c \wedge d)) \wedge (e \wedge f)$ would use only 3 extra bits. In principle this can be implemented by representing products as sets of variables, but in practice we found this greatly complicated the task of verification due to the more complicated data structure, while only saving at most $\log_2(k)$ bits.

i. Growing expressions One key difference between our partial evaluator and the semantics of Figure 5 is that the compiler evaluates Boolean terms to Boolean expressions rather than locations directly. For instance, given a term $t_1 \wedge t_2$ the compiler reduces each sub-expression t_1 and t_2 to Boolean expressions rather than locations – in comparison, the formal semantic definition reduces both t_1 and t_2 to locations first. While the equivalence of these two methods is not verified in our implementation, it is nonetheless easy to observe its correctness since all side-effectual computation is carried out during the reduction of t_1 and t_2 to Boolean expressions.

This method of growing expressions allows larger Boolean expressions to be optimized and efficiently compiled to circuits. In particular, a straightforward implementation of the operational semantics would require every binary sub-expression to be computed using an ancilla. In reality, this is not necessary, as in the case of compiling $t_1 \oplus t_2$ where no ancillas are allocated. While the Boolean expression interpretation goes farther and expands *all* locations within a Boolean expression, the expression size grows exponentially and quickly becomes unmanageable – we found this was a good compromise between the number of bits used and scalability of compilation.

V. TYPE INFERENCE

While the definition of REVER as a partial evaluator streamlines both development and verification, it leaves one issue: compiling functions. Semantically a function is a value, so the interpreter has nothing to do. In reality we wrap the evaluator with a procedure that allocates locations for the function parameters, then proceeds to evaluate the body; however, to do so, the compiler needs to know how many locations to allocate.

Ideally, this would be handled by type annotations, but this proved to be unnatural and confusing as F# has no mechanism for syntax extensions or dependent types. Instead we took the approach of using *type inference* together with a type system based on fixed-length registers. One of the practical benefits of our inference algorithm is it functions as both optimization and verification by only allocating bits that are actually used in the program. In particular, the inference algorithm exposed an off-by-one error in a REVS program by allocating fewer bits than were required by the algorithm.

We note that many other solutions are possible: the original REVS compiler for instance had programmers allocate inputs manually rather than write functions. This led to a significantly more complicated semantic model and somewhat unnatural looking programs. On the other hand, the problem could be avoided by delaying bit allocation until after the circuit is compiled. We opted for the type inference approach as it simplified verification, removing named variables from our internal representations which complicate formal verification [28].

j. Type system Our type system includes three basic types – the unit type, Booleans, and fixed-length registers – as well as the function type. As expected, indexing registers beyond their length causes a type error. To regain some polymorphism and flexibility the type system allows (structural) subtyping, in particular so that registers with greater size may be used anywhere a register with lesser size is required.

Type inference in systems with (structural) subtyping is generally considered a difficult problem – see, for e.g., [19] which proves NP-completeness of the problem for a similar type system involving record concatenation. While many attempts have been made ([20, 39–42]), they are typically not effective in practice and to the authors’ knowledge no such algorithms are currently in common use. Given the simplicity of our type system (in particular, the lack of non-subtype polymorphism), we have found a relatively simple inference algorithm based on constraint generation and *bound maps* is effective in practice.

Figure 6 summarizes algorithmic rules of our type system, which specify a set of constraints that any valid typing must satisfy. Constraints are given over a language of type and integer expressions. Type expressions include the basic types of REVER, as well variables representing types and registers over integer expressions.

$$\begin{array}{l}
\mathbf{IExp} \ \mathcal{I} ::= i \in \mathbb{N} \mid x \mid \mathcal{I}_1 \pm \mathcal{I}_2 \\
\mathbf{TExp} \ \mathcal{T} ::= X \mid \text{Unit} \mid \text{Bool} \mid \text{Register } \mathcal{I} \mid \mathcal{T}_1 \rightarrow \mathcal{T}_2 \\
\mathbf{Const} \ c ::= \mathcal{I}_1 = \mathcal{I}_2 \mid \mathcal{I}_1 \sqsubseteq \mathcal{I}_2 \mid \mathcal{T}_1 = \mathcal{T}_2 \mid \mathcal{T}_1 \sqsubseteq \mathcal{T}_2
\end{array}$$

$$\begin{array}{c}
\text{[C-LET]} \frac{\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \quad \Gamma, x : \mathcal{T}_1 \vdash t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_2}{\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_1 \cup \mathcal{C}_2} \\
\text{[C-VAR]} \frac{(x : \mathcal{T}) \in \Gamma}{\Gamma \vdash x : \mathcal{T} \downarrow \emptyset} \quad \text{[C-LAMBDA]} \frac{\Gamma, x : X \vdash t : \mathcal{T} \downarrow \mathcal{C} \quad X \text{ fresh}}{\Gamma \vdash \lambda x.t : X \rightarrow \mathcal{T} \downarrow \mathcal{C}} \\
\text{[C-APP]} \frac{\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \quad \Gamma \vdash t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_2 \quad \mathcal{C}_3 = \{\mathcal{T}_1 = X_1 \rightarrow X_2, X_2 \sqsubseteq \mathcal{T}_2\} \quad X_1, X_2 \text{ fresh}}{\Gamma \vdash (t_1 t_2) : X_2 \downarrow \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3} \quad \text{[C-SEQ]} \frac{\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \quad \Gamma \vdash t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_2}{\Gamma \vdash t_1; t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathcal{T}_1 = \text{Unit}\}} \\
\text{[C-BEXP]} \frac{\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \quad \Gamma \vdash t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_2}{\Gamma \vdash t_1 \star t_2 : \text{Bool} \downarrow \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathcal{T}_1 = \text{Bool}, \mathcal{T}_2 = \text{Bool}\}} \quad \text{[C-TRUE]} \frac{}{\Gamma \vdash \text{true} : \text{Bool} \downarrow \emptyset} \\
\text{[C-FALSE]} \frac{}{\Gamma \vdash \text{false} : \text{Bool} \downarrow \emptyset} \\
\text{[C-ASSN]} \frac{\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \quad \Gamma \vdash t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_2}{\Gamma \vdash t_1 \leftarrow t_2 : \text{Unit} \downarrow \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\mathcal{T}_1 = \text{Bool}, \mathcal{T}_2 = \text{Bool}\}} \\
\text{[C-APPEND]} \frac{\Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \quad \Gamma \vdash t_2 : \mathcal{T}_2 \downarrow \mathcal{C}_2 \quad \mathcal{C}_3 = \{\mathcal{T}_1 = \text{Register } x_1, \mathcal{T}_2 = \text{Register } x_2, x_3 = x_1 + x_2\} \quad x_1, x_2, x_3 \text{ fresh}}{\Gamma \vdash \text{append } t_1 t_2 : \text{Register } x_3 \downarrow \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3} \\
\text{[C-INDEX]} \frac{\Gamma \vdash t : \mathcal{T} \downarrow \mathcal{C}_1 \quad \mathcal{C}_2 = \{\mathcal{T} = \text{Register } x, i \sqsubseteq x\} \quad x \text{ fresh}}{\Gamma \vdash t.[i] : \text{Bool} \downarrow \mathcal{C}_1 \cup \mathcal{C}_2} \quad \Gamma \vdash t_1 : \mathcal{T}_1 \downarrow \mathcal{C}_1 \\
\vdots \\
\Gamma \vdash t_n : \mathcal{T}_n \downarrow \mathcal{C}_n \\
\text{[C-SLICE]} \frac{\Gamma \vdash t : \mathcal{T} \downarrow \mathcal{C}_1 \quad \mathcal{C}_2 \{\mathcal{T} = \text{Register } x, i \sqsubseteq j, j \sqsubseteq x\} \quad x \text{ fresh}}{\Gamma \vdash t.[i..j] : \text{Register } j - i + 1 \downarrow \mathcal{C}_1 \cup \mathcal{C}_2} \quad \text{[C-REG]} \frac{\mathcal{C}_{n+1} = \{\mathcal{T}_i = \text{Bool} \mid \forall 1 \leq i \leq n\}}{\Gamma \vdash \text{register } t_1 \dots t_n : \text{Register } n \downarrow \bigcup_{i=1}^{n+1} \mathcal{C}_i} \\
\text{[C-ROTATE]} \frac{\Gamma \vdash t : \mathcal{T} \downarrow \mathcal{C}_1 \quad \mathcal{C}_2 \{\mathcal{T} = \text{Register } x, i \sqsubseteq x\} \quad x \text{ fresh}}{\Gamma \vdash \text{rotate } t \ i : \text{Register } x \downarrow \mathcal{C}_1 \cup \mathcal{C}_2} \quad \text{[C-CLEAN]} \frac{\Gamma \vdash t : \mathcal{T} \downarrow \mathcal{C}}{\Gamma \vdash \text{clean } t : \text{Unit} \downarrow \mathcal{C} \cup \{\mathcal{T} = \text{Bool}\}} \\
\text{[C-ASSERT]} \frac{\Gamma \vdash t : \mathcal{T} \downarrow \mathcal{C}}{\Gamma \vdash \text{assert } t : \text{Unit} \downarrow \mathcal{C} \cup \{\mathcal{T} = \text{Bool}\}}
\end{array}$$

FIG. 6. Constraint typing rules

Integer expressions are linear expressions over integer-valued variables and constants. Equality and order relations are defined on type expressions as expected, while the integer expression ordering corresponds to the *reverse* order on integers (\geq) – this definition is used so that the maximal solution to an integer constraint gives the smallest register size. Constraints may be equalities or order relations between expressions of the same kind.

A *type substitution* Θ is a mapping from type variables to closed type expressions and integer variables to integers. By an abuse of notation we denote by $\Theta(\mathcal{T})$ the type \mathcal{T} with all variables replaced by their values in Θ . Additionally, we say that Θ satisfies the set of constraints \mathcal{C} if every constraint is true after applying the substitutions. The relation $\Gamma \vdash t : \mathcal{T} \downarrow \mathcal{C}$ means t can be given type $\Theta(\mathcal{T})$ for any type substitution satisfying \mathcal{C} .

k. Constraint solving We now give a brief sketch of our constraint solving algorithm, which finds a type substitution satisfying a set of constraints \mathcal{C} . We use a combination of unification (see, for e.g., [43]) and sets of upper and lower bounds for variables, similar to the method used in [20]. The function `computeBounds`, shown in pseudo-code in Algorithm 2, iterates through a set of constraints performing unification and computing the closure of the constraints wherever possible. Bounds – both subtype and equality – on variables are translated to a range of type or integer expressions, possibly open at one end, and the algorithm maintains a set of such bounds for each variable. To reduce complex linear arithmetic constraints to a variable bound we use a normalization procedure to write the constraint with a single positive polarity variable on one side.

After all constraints have translated to bounds we iterate through the variables, simplifying and checking

the set of upper and lower bounds. Any variable with no unassigned variable in its upper and lower bound sets is assigned the maximum value in the intersection of all its bounds; this process repeats until no variables are left unassigned. It is easy to observe that this algorithm terminates as the syntax of REVS does not allow circular systems of constraints to be generated. This has been confirmed in practice, as the constraint solver has terminated with the correct maximal bounds (or a type error) for every REVS program we tested.

Algorithm 2 Computation of bound sets.

```

function COMPUTEBOUNDS( $\mathcal{C}$ ,  $\theta$ )
   $\mathcal{C}$  is a set of constraints
   $\theta$  a set of (possible open at one end) ranges for each variable
  while  $c \in \mathcal{C}$  do
    if  $c$  is  $\mathcal{T}_1 = \mathcal{T}_2$  or  $\mathcal{I}_1 = \mathcal{I}_2$  then use unification
    else if  $c$  is  $\mathcal{T} \sqsubseteq \mathcal{T}$  or  $\mathcal{I} \sqsubseteq \mathcal{I}$  then
      computeBounds( $S \setminus \{c\}$ ,  $\theta$ )
    else if  $c$  is  $X \sqsubseteq \text{Unit}$  or  $\text{Unit} \sqsubseteq X$  then
      computeBounds( $S \setminus \{c\}$ ,  $\theta \cup \{X \mapsto [\text{Unit}, \text{Unit}]\}$ )
    else if  $c$  is  $X \sqsubseteq \text{Bool}$  or  $\text{Bool} \sqsubseteq X$  then
      computeBounds( $S \setminus \{c\}$ ,  $\theta \cup \{X \mapsto [\text{Bool}, \text{Bool}]\}$ )
    else if  $c$  is  $X \sqsubseteq \text{Register } \mathcal{I}$  then
      computeBounds( $S \setminus \{c\} \cup \{\mathcal{I} \sqsubseteq x\}$ ,  $\theta \cup \theta'$ )
      where  $\theta' = \{X \mapsto [\text{Register } x, \text{Register } x], x \mapsto [\mathcal{I}, \infty]\}$ 
    else if  $c$  is  $\text{Register } \mathcal{I} \sqsubseteq X$  then
      computeBounds( $S \setminus \{c\} \cup \{x \sqsubseteq \mathcal{I}\}$ ,  $\theta \cup \theta'$ )
      where  $\theta' = \{X \mapsto [\text{Register } x, \text{Register } x], x \mapsto [0, \mathcal{I}]\}$ 
    else if  $c$  is  $X \sqsubseteq \mathcal{T}_1 \rightarrow \mathcal{T}_2$  then
      computeBounds( $S \setminus \{c\} \cup \{\mathcal{T}_1 \sqsubseteq \mathcal{T}'_1, \mathcal{T}_2 \sqsubseteq \mathcal{T}'_2\}$ ,  $\theta \cup \theta'$ )
      where  $\theta' = \{X \mapsto [\mathcal{T}'_1 \rightarrow \mathcal{T}'_2, \mathcal{T}'_1 \rightarrow \mathcal{T}'_2]\}$ 
    else if  $c$  is  $\mathcal{T}_1 \rightarrow \mathcal{T}_2 \sqsubseteq X$  then
      computeBounds( $S \setminus \{c\} \cup \{\mathcal{T}'_1 \sqsubseteq \mathcal{T}_1, \mathcal{T}_2 \sqsubseteq \mathcal{T}'_2\}$ ,  $\theta \cup \theta'$ )
      where  $\theta' = \{X \mapsto [\mathcal{T}'_1 \rightarrow \mathcal{T}'_2, \mathcal{T}'_1 \rightarrow \mathcal{T}'_2]\}$ 
    else if  $c$  is  $X_1 \sqsubseteq X_2$  then
      computeBounds( $S \setminus \{c\}$ ,  $\theta \cup \{X_1 \mapsto [-, X_2]\}$ )
    else if  $c$  is  $x \sqsubseteq \mathcal{I}$  then
      computeBounds( $S \setminus \{c\}$ ,  $\theta \cup \{x \mapsto [-, \mathcal{I}]\}$ )
    else if  $c$  is  $\mathcal{I} \sqsubseteq x$  then
      computeBounds( $S \setminus \{c\}$ ,  $\theta \cup \{x \mapsto [\mathcal{I}, -]\}$ )
    else Fail
    end if
  end while
end function

```

VI. VERIFICATION

In this section we describe the verification of REVER and give the major theorems proven. All theorems given in this section have been formally specified and proven using the F* compiler [18]. We first give theorems about our Boolean expression compiler, then use these to prove properties about whole program compilation. The total verification of REVER comprises around 1500 lines of code.

Rather than give our proof code, we translate our proofs to more natural mathematical language as we believe this is more enlightening. The original theorems and proofs are given in Appendix A for reference.

A. Boolean expression compilation

1. *Correctness* Below is the main theorem establishing the correctness of compile-BExp with respect to the semantics of reversible circuits and Boolean expressions. It states that if the variables of B , the bits on

the ancilla heap and the target are non-overlapping, then the circuit computes the function $i \oplus B$.

Theorem 1. *Let B be a Boolean expression, ξ be the global ancilla heap, $i \in \mathbb{N}$ and s be a map from bits to Boolean values. Suppose $\text{vars}(B)$, ξ and $\{i\}$ are all disjoint and $s(j) = 0$ for all $j \in \xi$. Then*

$$s'(i) = s(i) \oplus \llbracket B \rrbracket s$$

where $s' = \llbracket \text{compile-BExp}(B, i) \rrbracket s$.

Proof. We use structural induction on B . We only sketch a recursive case, as the nonrecursive cases are trivial.

Consider the case $B = B_1 \oplus B_2$. Since $\text{vars}(B_1)$ is clearly disjoint with ξ and $\{i\}$, we see by induction that $s_1(i) = s(i) \oplus \llbracket B_1 \rrbracket s$ where $s_1 = \llbracket \text{compile-BExp}(B_1, i) \rrbracket s$. Next we want to use induction to show that $s_2(i) = s_1(i) \oplus \llbracket B_2 \rrbracket s_1$, where $s_2 = \llbracket \text{compile-BExp}(B_2, i) \rrbracket s_1$. To do so we observe that since no new ancillas are added to ξ by $\text{compile-BExp}(B_1, i)$, $\text{vars}(B_2)$, ξ and $\{i\}$ remain disjoint. Furthermore, since the circuit $\text{compile-BExp}(B_1, i)$ does not modify any bits still on the heap, $s_1(j) = 0$ for all $j \in \xi$ and hence $s_2 = \llbracket \text{compile-BExp}(B_2, i) \rrbracket s_1$ by induction.

Finally by the definition of $\text{compile-BExp}(B_1 \oplus B_2, i)$ and the semantics of reversible circuits, $s_2 = s'$. The \wedge case is slightly more involved, but mostly follows from the same argument. \square

m. Cleanup As remarked earlier, a crucial part of reversible computing is cleaning ancillas both to reduce space usage, and in quantum computing to prevent entangled qubits from influencing the computation. Moreover, the correctness of our cleanup is actually necessary to prove correctness of the compiler, as the compiler re-uses cleaned ancillas on the heap, potentially interfering with the precondition of Theorem 1. We use the following theorem to establish the correctness of our cleanup method, stating that the uncompute transformation reverses all changes on bits not in the target set under the condition that no bits in the target set are used as controls.

Theorem 2. *Let C be a well-formed reversible circuit and $A \subset \mathbb{N}$ be some set of bits. If $A \cap \text{control}(C) = \emptyset$ then for all states s , $s' = \llbracket C :: \text{uncompute}(C, A) \rrbracket s$ and any $i \notin A$,*

$$s(i) = s'(i)$$

Theorem 2 largely relies on the below lemma, which can be verified by a simple inductive case analysis:

Lemma 1. *Let $A \subset \mathbb{N}$ and s, s' be states such that for all $i \in A$, $s(i) = s'(i)$. If C is a reversible circuit where $\text{control}(C) \subseteq A$, then $(\llbracket C \rrbracket s)(i) = (\llbracket C \rrbracket s')(i)$ for all $i \in A$.*

Proof of Theorem 2. The proof proceeds by induction on the length of C . The base case is trivial, so we consider $C = C_1 :: C_2$. Without loss of generality, we assume C_1 is a single gate. If C_1 doesn't use any bit in B , then $\text{uncompute}(C, A) = \text{uncompute}(C_2, A) :: C_1$, so we use the inductive hypothesis on C_2 and the fact that C_1 is self-inverse.

On the other hand, if C_1 uses some $i \in A$, then i must be the target of C_1 . Then $\text{uncompute}(C, A) = \text{uncompute}(C_2, A)$, so we need to show that $s(i) = s'(i)$ where $s' = \llbracket C_1 :: C_2 :: \text{uncompute}(C_2, A) \rrbracket s$ for all $i \notin A$. We observe $s(i) = s'(i)$ where $s' = \llbracket C_1 \rrbracket s$ for all $i \notin A$, so by Lemma 1 we have

$$\llbracket C_1 :: C_2 :: \text{uncompute}(C_2, A) \rrbracket s(i) = \llbracket C_2 :: \text{uncompute}(C_2, A) \rrbracket s(i)$$

for all $i \notin A$. Thus by the inductive hypothesis we see that $s(i) = s'(i)$ where $s' = \llbracket C_1 :: C_2 :: \text{uncompute}(C_2, A) \rrbracket s$ for all $i \notin A$, as required. \square

Theorem 2, together with the fact that compile-BExp produces a well-formed circuit under disjointness constraints, gives us the corollary below that Boolean expression compilation with cleanup correctly reverses the changes to every bit except the target.

Corollary 1. *Let B be a Boolean expression, ξ be a the global ancilla heap and $i \in \mathbb{N}$ such that $\text{vars}(B)$, ξ and $\{i\}$ are all disjoint. Suppose $\text{compile-BExp}(B, i) = C$. Then for all $j \neq i$ and states $s, s' = \llbracket C \circ \text{uncompute}(C, \{i\}) \rrbracket s$ we have*

$$s'(j) = s(j)$$

n. Optimizations Due to lack of space we omit the correctness proofs of our optimization passes, but note that we prove correctness of our Boolean expression simplifications, expansion to ESOP form, and factoring of B as $i \oplus B'$.

B. REVS compilation

It was noted in Section IV that the design of REVER as a partial evaluator simplifies proving correctness. We expand on that point now, and in particular show that if a relation between the states of two interpretations is preserved by assignment, then the evaluator also preserves the relation. We state this formally in the theorem below.

Theorem 3. *Let $\mathcal{I}_1, \mathcal{I}_2$ be interpretations and suppose whenever $(\sigma_1, \sigma_2) \in R$ for some relation $R \subseteq \mathcal{I}_1 \times \mathcal{I}_2$, $(\text{assign}_1(\sigma_1, l, B), \text{assign}_2(\sigma_2, l, B)) \in R$ for any l, B . Then for any term t , if $\langle t, \sigma_1 \rangle \Rightarrow_{\mathcal{I}_1} \langle v, \sigma'_1 \rangle$ and $\langle t, \sigma_2 \rangle \Rightarrow_{\mathcal{I}_2} \langle v, \sigma'_2 \rangle$, then $(\sigma'_1, \sigma'_2) \in R$.*

Theorem 3 lifts properties about interpretations to properties of evaluation over those abstract machines – in particular, we only need to establish that *assignment* is correct for an interpretation to establish correctness of the corresponding evaluator/compiler. In practice this significantly reduces boilerplate proof code, which is useful as F* currently has limited support for automated induction.

Given two interpretations $\mathcal{I}, \mathcal{I}'$, we say states σ and σ' of \mathcal{I} and \mathcal{I}' are *observationally equivalent* with respect to a set of initial values $s \in \mathbf{State}$ if for all $i \in \mathbb{N}$, $\text{eval}_{\mathcal{I}}(\sigma, i, s) = \text{eval}_{\mathcal{I}'}(\sigma', i, s)$. We say $\sigma \sim_s \sigma'$ if σ and σ' are observationally equivalent with respect to s . We establish the preservation of observational equivalence between the three interpretations implemented in REVER, $\mathcal{I}_{\text{standard}}, \mathcal{I}_{\text{BExp}}$ and $\mathcal{I}_{\text{circuit}}$, to prove the correctness of our compiler.

Theorem 4. *For all states σ, σ' of $\mathcal{I}_{\text{standard}}$ and $\mathcal{I}_{\text{BExp}}$, respectively, and for all $l \in \mathbb{N}, B \in \mathbf{BExp}, s \in \mathbf{State}$, if $\sigma \sim_s \sigma'$ then*

$$\text{assign}_{\text{standard}}(\sigma, l, B) \sim_s \text{assign}_{\text{BExp}}(\sigma', l, B)$$

Proof. Follows directly from the compositionality of $\llbracket B \rrbracket$. □

Theorem 5. *For all states σ, σ' of $\mathcal{I}_{\text{standard}}$ and $\mathcal{I}_{\text{circuit}}$, respectively, and for all $l \in \mathbb{N}, B \in \mathbf{BExp}, s \in \mathbf{State}$, if $\sigma \sim_s \sigma'$ and $s(i) = 0$ whenever $i \in \xi$, then*

$$\text{assign}_{\text{standard}}(\sigma, l, B) \sim_s \text{assign}_{\text{circuit}}(\sigma', l, B).$$

Moreover, the ancilla heap remains 0-filled.

Proof. The correctness of the circuit interpretation is more difficult, since there is an extra level of abstraction where locations get mapped to bits within the circuit. Moreover, we have to add the extra condition that the ancilla heap is zero-filled, as observational equivalence alone is not strong enough to prove the theorem. As a result we need to establish three conditions to prove the theorem:

- the ancilla heap remains zero-filled,
- l is mapped to a bit containing the value of B , and
- no other location is mapped to a modified bit.

The first condition follows from the fact that Boolean expression compilation cannot add any new bits to the heap, while the second condition follows from the fact that compiled circuits only modify the target and ancillas.

The final condition effectively follows from a lemma stating that $\llbracket B \rrbracket s$ is equivalent to $\llbracket B[i \in \text{vars}(B) \mapsto \rho(i)] \rrbracket (\llbracket C \rrbracket s)$ – i.e., evaluating the expression B gives the same result as running the circuit C then evaluating B by replacing every variable in B with the state of the corresponding bit. This can be proven by a straightforward induction on the structure of B . □

Benchmark	REVS			REVER		
	bits	gates	Toffolis	bits	gates	Toffolis
carryRippleAdder 32	127	187	90	128	277	60
carryRippleAdder 64	255	379	186	256	565	124
mult 32	130	6016	4032	128	6111	4032
mult 64	258	24320	16256	256	24511	16256
carryLookahead 32	169	399	112	169	503	120
carryLookahead 64	439	1116	322	439	1382	336
modAdd 32	65	188	62	65	189	62
modAdd 64	129	380	126	129	381	126
cucarroAdder 32	129	98	32	65	99	32
cucarroAdder 64	257	194	64	129	195	64
ma4	17	24	8	17	24	8
SHA-2 2 rounds	577	2374	782	577	2388	782
SHA-2 4 rounds	833	4832	1592	833	4860	1592
SHA-2 8 rounds	1345	7712	3408	1345	10392	3408
SHA-2 16 rounds	2594	10336	7712	2594	23472	7712
MD5 2 rounds	7841	53824	18240	4769	45440	18240

TABLE I. Bit & gate counts for both compilers in default mode. Instances where one compiler reports fewer bits or Toffoli gates are bolded.

C. Assertion checking

Along with the formal *compiler* verification, REVER provides additional *program* verification tools in the form of a binary decision diagram (BDD) based assertion checker. BDD-based verification of both classical and reversible circuits is a common technique and has been shown to be effective even on some large circuits [23]. Our compiler provides the option to perform a verification pass where the interpreter is run using BDDs to represent and store Boolean expressions. The interpreter checks that BDDs for asserted values are equal to true and cleaned variables have BDD false, or else an error is raised. REVER also provides translation validation by generating the BDD(s) for an entire program and checking their equivalence to the BDD(s) for the circuit outputs.

VII. EXPERIMENTS

We ran experiments to compare the ancilla usage, gate and Toffoli counts of circuits compiled by REVER to the original REVS compiler. We compiled circuits for various arithmetic and cryptographic functions written in REVS using both compiler’s regular and space-efficient modes and reported the results in Tables I and II, respectively. Experiments were run in Linux using 8GB of RAM – for the space efficient REVER mode, most benchmarks ran out of memory, so we only report a few small examples.

The results show that in their default modes, REVS and REVER are more-or-less evenly matched in terms of bit counts. In fact, REVER used *fewer* bits on average despite also being certifiably correct. Our type inference algorithm also found bits allocated but never used in some REVS programs, which may have led to a reduction of bits. While REVER typically used more total gates than REVS, the number of Toffoli gates was only slightly worse for the carry lookahead adder, and significantly better for the carry ripple adder. Since Toffoli gates are generally much more costly than Not and controlled-Not gates – at least 7 times as typical implementations use 7 CNOT gates *per* Toffoli [4, 44], or up to hundreds of times in most fault-tolerant architectures [44] – we feel these gate counts are reasonable.

For the space-efficient compilation scheme, REVER’s Boolean expression compilation consistently produced

Benchmark	REVS			REVER		
	bits	gates	Toffolis	bits	gates	Toffolis
carryRippleAdder 2	8	19	6	6	9	5
carryRippleAdder 4	14	25	12	13	42	32
carryRippleAdder 8	30	61	36	29	10014	9996
mult 2	9	16	12	11	529	473
modAdd 2	5	8	2	3	8	4
modAdd 4	9	20	6	14	533	510
cucarroAdder 2	9	8	2	6	13	6
cucarroAdder 4	17	14	4	14	78	66
cucarroAdder 8	33	26	8	30	16406	16386
ma4	17	24	8	16	12	12

TABLE II. Space-efficient compilation results. Instances where one compiler reports fewer bits or Toffoli gates are bolded.

circuits with fewer bits than REVS’ dependence graph based scheme, as seen in Table II. In the cases of mult and modAdd 4, REVER used more bits as the programs were written in *in-place* style, while the space-efficient mode of REVER is strictly *out-of-place*. However, REVER used significantly more gates – almost 5000 times more in the most extreme case. This is due to the exponential increase in the expression size caused by expanding Boolean expressions to ESOP form, and hence the number of gates also grows exponentially. Moreover, because of the exponential size ESOP transformation, REVER’s space-efficient compilation is not scalable, whereas REVS’ method is.

While the results show there is clearly room for optimization of gate counts, they appear consistent with other verified compilers (e.g., [25]) which take some performance hit when compared to unverified compilers. In particular, unverified compilers may use more aggressive optimizations due to the increased ease of implementation and the lack of a requirement to prove their correctness compared to certified compilers. In some cases, the optimizations are even known to not be correct in all possible cases, as in the case of fast arithmetic and some loop optimization passes in the GNU C Compiler [45]. We found instances where REVS’ optimizations appeared to violate correctness, resulting in a semantically different circuit compared to the default compilation mode.

VIII. CONCLUSION

In this paper we described our verified compiler for the REVS language, REVER. We formalized the semantics of REVS and introduced a type system together with an inference algorithm. Our method of compilation differs from the original REVS compiler by using partial evaluation over an interpretation of the heap to compile programs, forgoing the need to re-implement and verify bookkeeping code for every internal translation. We described REVER’s two current methods of compilation in this framework, the default direct-to-circuit method, and a space efficient method that compiles first to a Boolean expression(s), then to a circuit.

While REVER is verified in the sense that compiled circuits produce the same result as the program interpreter, as with any verified compiler project this is not the end of certification. The implementation of the interpreter may have subtle bugs, which ideally would be verified against a more straightforward adaptation of the semantics using a relational definition. Moreover, our type inference algorithm and compilation “wrapper” have not yet been formally verified. We intend to address these issues in the future, and to extend REVER to utilize dependence graphs as in [1]. A parallel line of research is to extend REVS with new features that would make writing functions over complex data types easier, such as those used in quantum walks [4].

With a formally verified reversible circuit compiler, algorithm designers can assert with increased confi-

dence the actual circuit-level cost of running their algorithm, allowing a better assessment of the value of quantum computers. Perhaps more importantly, they are afforded the guarantee that data does not remain entangled with the garbage ancillas, allowing them to be safely reused later. Such guarantees are crucial to the operation of quantum computers and will become increasingly important as quantum computers continue to scale to larger sizes.

-
- [1] Alex Parent, Martin Roetteler, and Krysta M Svore, “Reversible circuit compilation with space constraints,” arXiv preprint arXiv:1510.00377 (2015).
 - [2] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” *SIAM Journal on Computing* **26**, 1484–1509 (1997).
 - [3] Lov K. Grover, “A fast quantum mechanical algorithm for database search,” in *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing (STOC 1996)*, edited by Gary L. Miller (ACM, 1996) pp. 212–219.
 - [4] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information* (Cambridge University Press, Cambridge, UK, 2000).
 - [5] I. L. Markov, “Limits on fundamental limits to computation,” *Nature* **512**, 147–154 (2014).
 - [6] D. M. Miller, D. Maslov, and G. W. Dueck, “A transformation based algorithm for reversible logic synthesis,” in *Proceedings of the 40th Design Automation Conference (DAC’03)* (2003) pp. 318–323.
 - [7] D. Maslov, D. M. Miller, and G. W. Dueck, “Techniques for the synthesis of reversible Toffoli networks,” *ACM Transactions on Design Automation of Electronic Systems* **12**, 42 (2007).
 - [8] R. Wille and R. Drechsler, *Towards a Design Flow for Reversible Logic*. (Springer, 2010).
 - [9] A. Shafaei, M. Saeedi, and M. Pedram, “Reversible logic synthesis of k -input, m -output lookup tables,” in *In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE’13)* (2013) pp. 1235–1240.
 - [10] C.-C. Lin and N. K. Jha, “RMDDS: Reed-Muller decision diagram synthesis of reversible logic circuits,” *ACM Journal on Emerging Technologies in Computing Systems* **10**, 14 (2014).
 - [11] M. Saeedi and I. L. Markov, “Synthesis and optimization of reversible circuits - a survey,” *ACM Comput. Surv.* **45**, 21 (2013).
 - [12] T. Yokoyama and R. Glück, “A reversible programming language and its invertible self-interpreter,” in *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (ACM, 2007) pp. 144–153.
 - [13] R. Wille, S. Offermann, and R. Drechsler, “Syrec: A programming language for synthesis of reversible circuits,” in *Specification Design Languages (FDL 2010), 2010 Forum on* (2010) pp. 1–6.
 - [14] M. K. Thomsen, “A functional language for describing reversible logic,” in *Proc. Forum on Specification and Design Languages (FDL’12)* (IEEE, 2012) pp. 135–142.
 - [15] A. S. Green, P. LeFanu Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, “Quipper: a scalable quantum programming language,” in *Proc. Conference on Programming Language Design and Implementation (PLDI’13)* (ACM, 2013).
 - [16] K. S. Perumalla, *Introduction to Reversible Computing* (CRC Press, 2014).
 - [17] A. Scherer, B. Valiron, S.-C. Mau, S. Alexander, E. van den Berg, and T. E. Chapuran, “Resource analysis of the quantum linear system algorithm,” ArXiv e-prints, arXiv:1505.06552 (2015), arXiv:1505.06552 [quant-ph].
 - [18] Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, and Jean-Karim Zinzindohoue, “Dependent types and multi-monadic effects in F*,” Draft (2015).
 - [19] Jens Palsberg and Tian Zhao, “Type inference for record concatenation and subtyping,” *Information and Computation* **189**, 54 – 86 (2004).
 - [20] Valery Trifonov and Scott Smith, “Subtyping constrained types,” in *Static Analysis, Lecture Notes in Computer Science*, Vol. 1145, edited by Radhia Cousot and David A. Schmidt (Springer Berlin Heidelberg, 1996) pp. 349–365.
 - [21] B. Ömer, *Quantum programming in QCL*, Master’s thesis, Technical University of Vienna (2000).
 - [22] Carlin Vieri, “Pendulum: A reversible computer architecture,” Master’s thesis, MIT Artificial Intelligence Laboratory (1995).
 - [23] R. Wille, D. Grosse, D.M. Miller, and R. Drechsler, “Equivalence checking of reversible circuits,” in *Multiple-Valued Logic, 2009. ISMVL ’09. 39th International Symposium on* (2009) pp. 324–330.
 - [24] S. Yamashita and I.L. Markov, “Fast equivalence-checking for quantum circuits,” in *Nanoscale Architectures (NANOARCH), 2010 IEEE/ACM International Symposium on* (2010) pp. 23–28.

- [25] Xavier Leroy, “Formal certification of a compiler back-end or: Programming a compiler with a proof assistant,” in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’06 (ACM, New York, NY, USA, 2006) pp. 42–54.
- [26] Xavier Leroy, “Formal verification of a realistic compiler,” *Commun. ACM* **52**, 107–115 (2009).
- [27] Lennart Beringer, Gordon Stewart, Robert Dockins, and AndrewW. Appel, “Verified compilation for shared-memory c,” in *Programming Languages and Systems*, Lecture Notes in Computer Science, Vol. 8410, edited by Zhong Shao (Springer Berlin Heidelberg, 2014) pp. 107–127.
- [28] Adam Chlipala, “A verified compiler for an impure functional language,” in *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’10 (ACM, New York, NY, USA, 2010) pp. 93–106.
- [29] Cédric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits, “Fully abstract compilation to JavaScript,” in *40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2013).
- [30] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens, “Cakeml: A verified implementation of ml,” in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’14 (ACM, New York, NY, USA, 2014) pp. 179–191.
- [31] JamesT. Perconti and Amal Ahmed, “Verifying an open compiler using multi-language semantics,” in *Programming Languages and Systems*, Lecture Notes in Computer Science, Vol. 8410, edited by Zhong Shao (Springer Berlin Heidelberg, 2014) pp. 128–148.
- [32] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis, “Pilsner: A compositionally verified compiler for a higher-order imperative language,” in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015 (ACM, New York, NY, USA, 2015) pp. 166–178.
- [33] C. H. Bennett, “Logical reversibility of computation,” *IBM Journal of Research and Development* **17**, 525–532 (1973).
- [34] A. S. Green, P. LeFanu Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, “An introduction to quantum programming in Quipper,” in *Proc. Reversible Computation (RC’13)* (ACM, 2013).
- [35] Koen Claessen, *Embedded Languages for Describing and Verifying Hardware*, Phd thesis, Chalmers University of Technology and Göteborg University (2001).
- [36] NIST, “Federal information processing standards publication 180-2,” (2002), see also the Wikipedia entry <http://en.wikipedia.org/wiki/SHA-2>.
- [37] C. H. Bennett, “Time/space trade-offs for reversible computation,” *SIAM Journal on Computing* **18**, 766–776 (1989).
- [38] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft, *Partial Evaluation and Automatic Program Generation* (Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993).
- [39] John C. Mitchell, “Type inference with simple subtypes,” *Journal of Functional Programming* **1**, 245–285 (1991).
- [40] Jonathan Eifrig, Scott Smith, and Valery Trifonov, “Sound polymorphic type inference for objects,” in *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA ’95 (ACM, New York, NY, USA, 1995) pp. 169–184.
- [41] François Pottier, “Simplifying subtyping constraints,” in *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP ’96 (ACM, New York, NY, USA, 1996) pp. 122–133.
- [42] Martin Odersky, Martin Sulzmann, and Martin Wehr, “Type inference with constrained types,” *Theory and Practice of Object Systems* **5**, 35–55 (1999).
- [43] Luis Damas and Robin Milner, “Principal type-schemes for functional programs,” in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’82 (ACM, New York, NY, USA, 1982) pp. 207–212.
- [44] M. Amy, D. Maslov, M. Mosca, and M. Roetteler, “A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits,” *Computer-Aided Design of Integrated Circuits and Systems*, *IEEE Transactions on* **32**, 818–830 (2013).
- [45] “Using the GNU Compiler Collection,” Free Software Foundation, Inc. (2016).

Appendix A: Proof listings

In this appendix we give the F* source code listings for the theorems and proofs given in Section VI. As we do not include auxiliary definitions and theorems, the code is included for reference against the less formal mathematical statements given in the main text.

```

1 (* Theorem 1 *)
2 val compile_bexp_correct : ah:AncHeap -> targ:int -> exp:BExp -> st:state ->
3   Lemma (requires (zeroHeap st ah /\ disjoint (elts ah) (vars exp) /\
4     not (Util.mem targ (elts ah)) /\
5     not (Util.mem targ (vars exp))))
6     (ensures (compileBExpEval ah targ exp st = st targ <> evalBExp exp st))
7   (decreases %[exp;0])
8 val compile_bexp_correct_ooop : ah:AncHeap -> exp:BExp -> st:state ->
9   Lemma (requires (zeroHeap st ah /\ disjoint (elts ah) (vars exp)))
10    (ensures (compileBExpEval_ooop ah exp st = evalBExp exp st))
11    (decreases %[exp;1])
12 let rec compile_bexp_correct ah targ exp st = match exp with
13 | BFalse -> ()
14 | BVar x -> ()
15 | BNot x ->
16   let (ah', xres, xanc, xgate) = compileBExp ah targ x in
17   let ind_hyp_x = compile_bexp_correct ah targ x st in
18   evalCirc_append xgate [RNOT xres] st
19 | BXor (x, y) ->
20   let (ah', xres, xanc, xgate) = compileBExp ah targ x in
21   let (ah'', yres, yanc, ygate) = compileBExp ah' targ y in
22   let st' = evalCirc xgate st in
23   let ind_hyp_x = compile_bexp_correct ah targ x st in
24   let ind_hyp_y =
25     compile_decreases_heap ah targ x;
26     disjoint_subset (elts ah') (elts ah) (vars y);
27     compile_partition ah targ x;
28     zeroHeap_subset st ah ah';
29     zeroHeap_st_impl st ah' xgate;
30     compile_bexp_correct ah' targ y st'
31   in
32   let lem1 = (* (eval (xgate@ygate)) targ = (eval ygate st') targ *)
33     evalCirc_append xgate ygate st
34   in
35   let lem2 = (* eval y st = eval y st' *)
36     compile_mods ah targ x;
37     eval_mod st xgate;
38     eval_state_swap y st st'
39   in ()
40 | BAnd (x, y) ->
41   let (ah', xres, xanc, xgate) = compileBExp_ooop ah x in
42   let (ah'', yres, yanc, ygate) = compileBExp_ooop ah' y in
43   let st' = evalCirc xgate st in
44   let ind_hyp_x = compile_bexp_correct_ooop ah x st in
45   let ind_hyp_y =
46     compile_decreases_heap_ooop ah x;
47     disjoint_subset (elts ah') (elts ah) (vars y);
48     compile_partition_ooop ah x;
49     zeroHeap_subset st ah ah';
50     zeroHeap_st_impl st ah' xgate;
51     compile_bexp_correct_ooop ah' y st'
52   in
53   let lem1 = (* st' xres = (evalCirc ygate st') xres *)
54     compile_mods_ooop ah' y;
55     eval_mod st' ygate
56   in
57   let lem2 = (* eval y st = eval y st' *)
58     compile_mods_ooop ah x;

```

```

59     eval_mod st xgate;
60     eval_state_swap y st st'
61   in
62   let lem3 = () (* st targ = (evalCirc ygate st') targ *)
63   in
64     evalCirc_append xgate ygate st;
65     evalCirc_append (xgate@ygate) [RTOFF (xres, yres, targ)] st
66 and compile_bexp_correct_oop ah exp st = match exp with
67 | BVar v -> ()
68 | _ ->
69   let (ah', targ) = popMin ah in
70   let (ah'', _, _, gates) = compileBExp ah' targ exp in
71     pop_proper_subset ah;
72     pop_elt ah;
73     compile_bexp_correct ah' targ exp st

75 (* Lemma 1 *)
76 val evalCirc_state_lem : circ:list Gate -> st:state -> st':state -> dom:set int ->
77   Lemma (requires (subset (ctrls circ) dom /\ agree_on st st' dom))
78     (ensures (agree_on (evalCirc circ st) (evalCirc circ st') dom))
79 let rec evalCirc_state_lem circ st st' dom = match circ with
80 | [] -> ()
81 | x::xs ->
82   applyGate_state_lem x st st' dom;
83   evalCirc_state_lem xs (applyGate st x) (applyGate st' x) dom

85 val uncompute_agree : circ:list Gate -> dom:set int -> st:state ->
86   Lemma (requires (wfCirc circ /\ disjoint (ctrls circ) dom))
87     (ensures (agree_on (evalCirc circ st)
88                       (evalCirc (uncompute circ dom) st)
89                       (complement dom)))
90 let rec uncompute_agree circ dom st = match circ with
91 | [] -> ()
92 | x::xs ->
93   if (contains_lst dom (vars [x]))
94   then
95     (evalCirc_state_lem xs (applyGate st x) st (complement dom);
96     uncompute_agree xs targ st;
97     agree_on_trans (evalCirc xs (applyGate st x))
98                   (evalCirc xs st)
99                   (evalCirc (uncompute xs dom) st)
100                  (complement dom))
101   else uncompute_agree xs dom (applyGate st x)

103 (* Theorem 2 *)
104 val uncompute_correct : circ:list Gate -> dom:set int -> st:state ->
105   Lemma (requires (wfCirc circ /\ disjoint (ctrls circ) dom))
106     (ensures (agree_on st (evalCirc (List.rev (uncompute circ dom))
107                                   (evalCirc circ st))
108              (complement dom)))
109 let uncompute_correct circ dom st =
110   uncompute_agree circ dom st;
111   uncompute_ctrls_subset circ dom;
112   evalCirc_state_lem (List.rev (uncompute circ dom))
113                     (evalCirc circ st)
114                     (evalCirc (uncompute circ dom) st)
115                     (complement dom);
116   rev_inverse (uncompute circ dom) st

118 (* Corollary 1 *)
119 val compile_with_cleanup : ah:AncHeap -> targ:int -> exp:BExp -> st:state ->
120   Lemma (requires (zeroHeap st ah /\ disjoint (elts ah) (vars exp) /\
121                 not (Util.mem targ (elts ah)) /\
122                 not (Util.mem targ (vars exp))))
123     (ensures (clean_heap_cond ah targ exp st /\

```

```

124         clean_corr_cond ah targ exp st))
125 let compile_with_cleanup ah targ exp st =
126   let (ah', res, anc, circ) = compileBExp ah targ exp in
127   let cleanup = uncompute_circ (singleton res) in
128   let ah'' = List.fold_leftT insert ah' anc in
129   let st' = evalCirc circ st in
130   let st'' = evalCirc (circ@(List.rev cleanup)) st in
131   let heap_cond =
132     let leml = (* zeroHeap st' ah' *)
133       compile_decreases_heap ah targ exp;
134       compile_partition ah targ exp;
135       zeroHeap_subset st ah ah';
136       zeroHeap_st_impl st ah' circ
137     in
138     let leml = (* zeroHeap st'' ah' *)
139       compileBExp_wf ah targ exp;
140       uncompute_uses_subset circ (singleton res);
141       zeroHeap_st_impl st' ah' (List.rev cleanup)
142     in
143       compile_ctrls ah targ exp;
144       uncompute_correct circ (singleton res) st;
145       compile_anc ah targ exp;
146       zeroHeap_insert_list st'' ah' anc
147   in
148   let corr_cond =
149     uncompute_targ circ res;
150     eval_mod st' (List.rev cleanup)
151   in ()
152 val compile_with_cleanup_oop : ah:AncHeap -> exp:BExp -> st:state ->
153   Lemma (requires (zeroHeap st ah /\ disjoint (elts ah) (vars exp)))
154     (ensures (zeroHeap (compileBExpCleanEvalSt_oop ah exp st)
155       (first (compileBExpClean_oop ah exp)) /\
156         compileBExpCleanEval_oop ah exp st =
157         compileBExpEval_oop ah exp st))
158 let compile_with_cleanup_oop ah exp st =
159   let (ah', targ) = popMin ah in
160   compile_with_cleanup ah' targ exp st

162 (* Optimization theorems *)
163 val simplify_preserves_semantics : exp:BExp ->
164   Lemma (forall (st:state). (evalBExp exp st) = (evalBExp (simplify exp) st))
165 let rec simplify_preserves_semantics exp = match exp with
166 | BFalse -> ()
167 | BVar x -> ()
168 | BAnd (x, y) | BXor (x, y) ->
169   simplify_preserves_semantics x;
170   simplify_preserves_semantics y
171 | BNot x -> simplify_preserves_semantics x

173 val factorAs_correct : exp:BExp -> targ:int -> st:state ->
174   Lemma (forall exp'. factorAs exp targ = Some exp' ==>
175     not (occursInBExp targ exp') /\ evalBExp exp st = st targ <> evalBExp exp' st)
176 let rec factorAs_correct exp targ st = match exp with
177 | BFalse -> ()
178 | BVar x -> ()
179 | BNot x -> factorAs_correct x targ st
180 | BAnd (x, y) -> ()
181 | BXor (x, y) ->
182   factorAs_correct x targ st;
183   factorAs_correct y targ st

185 val dist_preserves_semantics : exp:BExp ->
186   Lemma (forall (st:state). (evalBExp exp st) = (evalBExp (distributeAnds exp) st))
187 let rec dist_preserves_semantics exp = match exp with
188 | BFalse -> ()

```

```

189 | BVar x -> ()
190 | BNot x -> dist_preserves_semantics x
191 | BXor (x, y) -> dist_preserves_semantics x; dist_preserves_semantics y
192 | BAnd (x, y) ->
193   dist_preserves_semantics x;
194   dist_preserves_semantics y;
195   begin match (distributeAnds x, distributeAnds y) with
196   | (BXor (a, b), BXor (c, d)) ->
197     distributivityAndXor (BXor (a, b)) c d;
198     commutativityAnd (BXor (a, b)) c;
199     commutativityAnd (BXor (a, b)) d;
200     distributivityAndXor c a b;
201     distributivityAndXor d a b;
202     commutativityAnd c a;
203     commutativityAnd c b;
204     commutativityAnd d a;
205     commutativityAnd d b
206   | (x', BXor (c, d)) -> distributivityAndXor x' c d
207   | (BXor (a, b), y') ->
208     commutativityAnd (BXor (a, b)) y';
209     distributivityAndXor y' a b;
210     commutativityAnd y' a;
211     commutativityAnd y' b
212   | (x', y') -> ()
213   end

215 (* Theorem 4 and related lemma *)
216 type state_equiv (st:boolState) (st':BExpState) (init:state) =
217   forall i. boolEval st init i = bexpEval st' init i

219 val eval_bexp_swap : st:boolState -> st':BExpState -> bexp:BExp -> init:state ->
220   Lemma (requires (state_equiv st st' init))
221     (ensures (evalBExp (substBExp bexp (snd st')) init =
222       evalBExp bexp (snd st)))
223 let rec eval_bexp_swap st st' bexp init = match bexp with
224 | BFalse -> ()
225 | BVar i -> ()
226 | BNot x -> eval_bexp_swap st st' x init
227 | BXor (x, y) | BAnd (x, y) ->
228   eval_bexp_swap st st' x init;
229   eval_bexp_swap st st' y init

231 val state_equiv_assign : st:boolState -> st':BExpState -> init:state -> l:int -> bexp:BExp ->
232   Lemma (requires (state_equiv st st' init))
233     (ensures (state_equiv (boolAssign st l bexp) (bexpAssign st' l bexp) init))
234 let state_equiv_assign st st' init l bexp = eval_bexp_swap st st' bexp init

236 (* Theorem 5 and related lemmas *)
237 type circ_equiv (st:boolState) (cs:circState) (init:state) =
238   zeroHeap (evalCirc cs.gates init) cs.ah /\
239   (forall i. not (mem (lookup cs.subs i) cs.ah)) /\
240   (forall i. boolEval st init i = circEval cs init i)

242 val eval_commutates_subst_circ : st:boolState -> cs:circState -> bexp:BExp ->
243   bexp':BExp -> init:state -> targ:int -> targ':int ->
244   Lemma (requires (circ_equiv st cs init /\
245     bexp' = substVar bexp cs.subs /\
246     targ' = lookup cs.subs targ /\
247     not (Util.mem targ' (vars bexp')) /\
248     not (Util.mem targ' (elts cs.ah)) /\
249     disjoint (elts cs.ah) (vars bexp')))
250     (ensures ((evalCirc (last (compileBExp cs.ah targ' bexp'))
251       (evalCirc cs.gates init)) targ' =
252       lookup (snd st) targ <> evalBExp bexp (snd st)))
253 let eval_commutates_subst_circ st cs bexp bexp' init targ targ' =

```

```

254   let init' = evalCirc cs.gates init in
255     compile_bexp_correct cs.ah targ' bexp' init';
256     eval_bexp_swap st cs bexp bexp' init

258 val eval_commutates_subst_circ_oop : st:boolState -> cs:circState ->
259   bexp:BExp -> bexp':BExp -> init:state ->
260   Lemma (requires (circ_equiv st cs init /\
261     bexp' = substVar bexp cs.subs /\
262     disjoint (elts cs.ah) (vars bexp')))
263     (ensures ((evalCirc (last (compileBExp_oop cs.ah bexp')))
264       (evalCirc cs.gates init))
265       (second (compileBExp_oop cs.ah bexp'))) =
266       evalBExp bexp (snd st)))
267 let eval_commutates_subst_circ_oop st cs bexp bexp' init =
268   let init' = evalCirc cs.gates init in
269     compile_bexp_correct_oop cs.ah bexp' init';
270     eval_bexp_swap st cs bexp bexp' init

272 val circ_equiv_assign : st:boolState -> cs:circState -> init:state -> l:int -> bexp:BExp ->
273   Lemma (requires (circ_equiv st cs init))
274     (ensures (circ_equiv (boolAssign st l bexp) (circAssign cs l bexp) init))
275 let circ_equiv_assign st cs init l bexp =
276   let l' = lookup cs.subs l in
277     let bexp' = substVar bexp cs.subs in
278     let init' = evalCirc cs.gates init in
279     let st' = boolAssign st l bexp in
280     let cs' = circAssign cs l bexp in
281     match factorAs bexp' l' with
282     | None ->
283       let (ah', res, ancs, circ') = compileBExp_oop cs.ah bexp' in
284       let zeroHeap_lem =
285         compile_decreases_heap_oop cs.ah bexp';
286         compile_partition_oop cs.ah bexp';
287         zeroHeap_subset init' cs.ah cs'.ah;
288         zeroHeap_st_impl init' cs'.ah circ'
289       in
290       let preservation =
291         compile_mods_oop cs.ah bexp';
292         eval_mod init' circ'
293       in
294       let correctness =
295         eval_commutates_subst_circ_oop st cs bexp bexp' init
296       in ()
297     | Some bexp'' ->
298       let (ah', res, ancs, circ') = compileBExp cs.ah l' bexp'' in
299       let zeroHeap_lem =
300         factorAs_correct bexp' l' init';
301         factorAs_vars bexp' l';
302         compile_decreases_heap cs.ah l' bexp'';
303         compile_partition cs.ah l' bexp'';
304         zeroHeap_subset init' cs.ah cs'.ah;
305         zeroHeap_st_impl init' cs'.ah circ'
306       in
307       let preservation =
308         compile_mods cs.ah l' bexp'';
309         eval_mod init' circ'
310       in
311       let correctness =
312         admitP(b2t(lookup (snd st') l = (evalCirc circ' init') (lookup cs'.subs l)));
313         factorAs_correct bexp' l' init';
314         eval_commutates_subst_circ st cs bexp bexp' init l l'
315       in ()

```