Verified compilation of space-efficient reversible circuits

Matthew Amy[1,2], Martin Roetteler[3], and Krysta M. Svore[3]

[1] Institute for Quantum Computing, Waterloo, Canada
[2] David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada
[3] Microsoft Research, Redmond, USA

Abstract. 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, a process that is crucial for identifying which algorithms will outperform their classical counterparts. We present a reversible circuit compiler called ReVerC, 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 [21] to combinational reversible circuits with as few ancillary bits as possible, and provably cleans temporary values.

1 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 [26] uses classical modular arithmetic and Grover's quantum algorithm [11] 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 [19]. Other applications of reversible computing include low-power design of classical circuits. See [15] 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 [14,16,17,25,29] (see also [23] for a survey) to high-level programming languages and compilers [10,21,28,31,33]. 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., [24]). The 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 exist 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 ReVerC, a lightly optimizing compiler for the Revs language [21] which has been written and proven correct in the dependently typed language F* [27]. Circuits compiled with ReVerC are certified 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, ReVerC provides an assertion checker which can be used to formally verify the source program itself, allowing effective end-to-end verification of reversible circuits.

Contributions The following is a summary of the contributions of our paper:

- We give a formal semantics of Revs.
- We present a compiler for Revs called ReVerC, written in F*. The compiler currently has three modes: direct to circuit, eager-cleaning, and Boolean expression compilation.
- We develop a new method of eagerly cleaning bits to be reused again later, based on *cleanup expressions*.
- Finally, we verify correctness of ReVerC 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.

Related work Due to the reversibility requirement of quantum computing, quantum programming languages and compilers typically have 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 [20], "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 [10] 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 [24].

Verification of reversible circuits has been previously considered from the viewpoint of checking equivalence against a benchmark circuit or specification [30, 32]. 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. Recent work has shown progress towards verification of more general properties of reversible and quantum circuits via model checking [4], but 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 [13], 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 [6], functional programming [7, 9] and modularity [18, 22].

2 Reversible computing

Reversible functions are Boolean functions $f : \{0,1\}^n \to \{0,1\}^n$ 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 [19].

An important metric that is associated with 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. Time flows left to right, with input values listed on the left and outputs listed on the right. NOT gates are depicted 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.

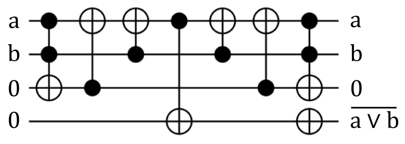

Fig. 1. A Toffoli network computing the NOR function $f(a,b) = \overline{a \vee b}$.

Revs From Bennett's work on reversible Turing machines it follows that any function can be implemented by a suitable reversible circuit [5]: 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 used to perform classical-to-reversible circuit compilation in the quantum programming language Quipper [10]. 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.

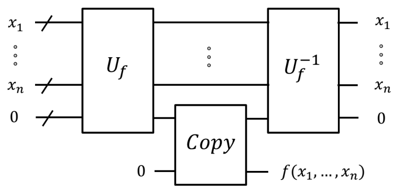

Fig. 2. A reversible circuit computing $f(x_1, \ldots, x_m)$ using the Bennett trick. Input lines with slashes denote an arbitrary number of bits.

In recent work, an attempt was made with the Revs compiler (and programming language of the same name) [21] 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. Their method makes use of a dependency graph to determine which bits may be eligible to be cleaned *eagerly*, before the end of the

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

Fig. 3. Syntax of Revs.

computation and hence be reused again. We build on their work in this paper, formalizing Revs and developing a *verified* compiler without too much loss in efficiency. In particular, we take the idea of eager cleanup and develop a new method analogous to garbage collection.

3 Languages

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

3.1 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* which asserts that its argument evaluates to 0 and frees a bit.

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

$$
\neg t \overset{\Delta}{=} 1 \oplus t, \qquad t_1 \vee t_2 \overset{\Delta}{=} (t_1 \wedge t_2) \oplus (t_1 \oplus t_2),
$$
$$
\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \overset{\Delta}{=} (t_1 \wedge t_2) \oplus (\neg t_1 \wedge t_3),
$$
$$
\mathsf{for}\ x\ \mathsf{in}\ i..j\ \mathsf{do}\ t \overset{\Delta}{=} t[x \mapsto i]; \cdots ; t[x \mapsto j].
$$

Note that Revs has no *dynamic* control – i.e. control dependent on run-time values. In particular, every Revs program can be transformed into a straight-line program. This is due to the restrictions of our target architecture (see below).

The ReVerC 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 [8]. We use F#'s *quotations* mechanism to achieve this by writing Revs programs in quotations `<@...@>`. Note that unlike languages such as Quipper, our strictly combinational target architecture doesn't allow computations in the meta-language to depend on computations within Revs.

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

 6    result.[0] ← a.[0] ⊕ b.[0]
 7    for i in 1 .. n-1 do
 8      carry ← carry_ex a.[i-1] b.[i-1] carry
 9      result.[i] ← a.[i] ⊕ b.[i] ⊕ carry
10    result
```

Fig. 4. Implementation of an n-bit adder.

Example 1. Figure 4 gives an example of a carry-ripple adder written in Revs. Naïvely compiling this implementation would result in a new bit being allocated for every carry bit, as the assignment on line 8 is irreversible (note that `carry_ex 1 1 0` $=$ `1` $=$ `carry_ex 1 1 1`, hence the value of `c` can not be uniquely computed given `a`, `b` and the output). ReVerC reduces this space usage by automatically cleaning the old carry bit, allowing it to be reused.

Semantics We designed the semantics of Revs with two goals in mind:

1. keep the semantics as close to the original implementation 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 semantics naturally enforces a style of programming that results in efficient circuits and allows common design patterns to be optimized.

The big-step semantics of Revs is presented in Figure 5 as a relation $\Rightarrow \subseteq \text{Config} \times \text{Config}$ on configuration-pairs – pairs of terms and Boolean-valued stores. A key feature of our semantics 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.

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* – i.e. arrays of bits. Specifically, registers are modelled as static length lists of bits. This allows the programmer to perform array-like operations such as bit modifications ($t_1.[i] \leftarrow t_2$) as well as list-like operations such as slicing ($t.[i..j]$) and concatenation ($\mathsf{append}\ t_1\ t_2$) without copying out entire registers. We found that these were the most common access patterns for arrays of bits in low-level bitwise code (e.g. arithmetic and cryptographic implementations).

The semantics of $\oplus$ (Boolean XOR) and $\wedge$ (Boolean AND) are also notable in that they first reduce both arguments to locations, *then* retrieve their value. This results in statements whose value may not be immediately apparent – e.g.,

$$\textbf{Store}\ \ \sigma : \mathbb{N} \rightharpoonup \mathbb{B}$$

$$\textbf{Config}\ \ c ::= \langle t, \sigma \rangle$$

$$[\textsc{let}]\ \dfrac{\langle t_1, \sigma \rangle \Rightarrow \langle v_1, \sigma' \rangle \qquad \langle t_2[x \mapsto v_1], \sigma' \rangle \Rightarrow \langle v_2, \sigma'' \rangle}{\langle \mathsf{let}\ x = t_1\ \mathsf{in}\ t_2, \sigma \rangle \Rightarrow \langle v_2, \sigma'' \rangle}$$

$$[\textsc{refl}]\ \dfrac{}{\langle v, \sigma \rangle \Rightarrow \langle v, \sigma \rangle}$$

$$[\textsc{bexp}]\ \dfrac{\langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma' \rangle \quad \langle t_2, \sigma' \rangle \Rightarrow \langle l_2, \sigma'' \rangle \quad l_3 \notin \mathsf{dom}(\sigma'')}{\langle t_1 \star t_2, \sigma \rangle \Rightarrow \langle l_3, \sigma''[l_3 \mapsto \sigma''(l_1) \star \sigma''(l_2)] \rangle}$$

$$[\textsc{bool}]\ \dfrac{b \in \mathbb{B} \quad l \notin \mathsf{dom}(\sigma)}{\langle b, \sigma \rangle \Rightarrow \langle l, \sigma''[l \mapsto b] \rangle}$$

$$[\textsc{app}]\ \dfrac{\begin{array}{c}\langle t_1, \sigma \rangle \Rightarrow \langle \lambda x.t_1', \sigma' \rangle \langle t_2, \sigma' \rangle \Rightarrow \langle v_2, \sigma'' \rangle \\ \langle t_1'[x \mapsto v_2], \sigma'' \rangle \Rightarrow \langle v, \sigma''' \rangle\end{array}}{\langle (t_1\ t_2), \sigma \rangle \Rightarrow \langle v, \sigma''' \rangle}$$

$$[\textsc{seq}]\ \dfrac{\begin{array}{c}\langle t_1, \sigma \rangle \Rightarrow \langle \mathsf{unit}, \sigma' \rangle \\ \langle t_2, \sigma' \rangle \Rightarrow \langle v, \sigma'' \rangle\end{array}}{\langle t_1; t_2, \sigma \rangle \Rightarrow \langle v, \sigma'' \rangle}$$

$$[\textsc{assn}]\ \dfrac{\begin{array}{c}\langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma' \rangle \\ \langle t_2, \sigma' \rangle \Rightarrow \langle l_2, \sigma'' \rangle\end{array}}{\langle t_1 \leftarrow t_2, \sigma \rangle \Rightarrow \langle \mathsf{unit}, \sigma''[l_1 \mapsto \sigma''(l_2)] \rangle}$$

$$[\textsc{append}]\ \dfrac{\langle t_1, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_1 \dots l_m, \sigma' \rangle \quad \langle t_2, \sigma' \rangle \Rightarrow \langle \mathsf{reg}\ l_{m+1} \dots l_n, \sigma'' \rangle}{\langle \mathsf{append}\ t_1\ t_2, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_1 \dots l_n, \sigma'' \rangle}$$

$$[\textsc{index}]\ \dfrac{\langle t, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_1 \dots l_n, \sigma' \rangle \qquad 1 \leq i \leq n}{\langle t.[i], \sigma \rangle \Rightarrow \langle l_i, \sigma' \rangle}$$

$$[\textsc{slice}]\ \dfrac{\langle t, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_1 \dots l_n, \sigma' \rangle \quad 1 \leq i \leq j \leq n}{\langle t.[i..j], \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_i \dots l_j, \sigma' \rangle}$$

$$[\textsc{reg}]\ \dfrac{\begin{array}{c}\langle t_1, \sigma \rangle \Rightarrow \langle l_1, \sigma_1 \rangle \\ \langle t_2, \sigma \rangle \Rightarrow \langle l_2, \sigma_2 \rangle \\ \vdots \\ \langle t_n, \sigma \rangle \Rightarrow \langle l_n, \sigma_n \rangle\end{array}}{\langle \mathsf{reg}\ t_1 \dots t_n, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_1 \dots l_n, \sigma_n \rangle}$$

$$[\textsc{rotate}]\ \dfrac{\langle t, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_1 \dots l_n, \sigma' \rangle \quad 1 < i < n}{\langle \mathsf{rotate}\ t\ i, \sigma \rangle \Rightarrow \langle \mathsf{reg}\ l_i \dots l_{i-1}, \sigma' \rangle}$$

$$[\textsc{clean}]\ \dfrac{\langle t, \sigma \rangle \Rightarrow \langle l, \sigma' \rangle \quad \sigma'(l) = 0}{\langle \mathsf{clean}\ t, \sigma \rangle \Rightarrow \langle \mathsf{unit}, \sigma'|_{\mathsf{dom}(\sigma') \setminus \{l\}} \rangle}$$

$$[\textsc{assert}]\ \dfrac{\langle t, \sigma \rangle \Rightarrow \langle l, \sigma' \rangle \quad \sigma'(l) = 1}{\langle \mathsf{assert}\ t, \sigma \rangle \Rightarrow \langle \mathsf{unit}, \sigma' \rangle}$$

Fig. 5. Operational semantics of Revs.

$x \oplus (x \leftarrow y; y)$, 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.

3.2 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.

A Boolean expression is defined as an expression over Boolean constants, variable indices, and logical $\oplus$ and $\wedge$ operators. Explicitly, we define

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

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

We interpret a Boolean expression as a function from (total) Boolean-valued states to Booleans. In particular, we define **State** $= \mathbb{N} \to \mathbb{B}$ and denote the semantics of a Boolean expression by $[\![B]\!] : \textbf{State} \to \mathbb{B}$. The formal definition of $[\![B]\!]$ is obvious so we omit it.

3.3 Target architecture

ReVerC compiles to *combinational, reversible circuits* over NOT, controlled-NOT and Toffoli gates. By combinational circuits we mean a sequence 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 [19].

Formally, we define

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

i.e., **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* and is the only bit *modified* or changed by the gate. We use $\mathsf{use}(C)$, $\mathsf{mod}(C)$ and $\mathsf{control}(C)$ 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 $[\![C]\!] : \textbf{State} \to \textbf{State}$, is straightforward:

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

We use $s[x \mapsto y]$ to denote the 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.

4 Compilation

In this section we discuss the implementation of ReVerC. The compiler consists of around 4000 lines of code in a common subset of F* and F#, with a front-end to evaluate and translate F# quotations into Revs expressions.

4.1 Boolean expression compilation

The core of ReVerC's code generation is a compiler from Boolean expressions into reversible circuits. We use a modification of the method employed in Revs.

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. ReVerC does this by maintaining a (mutable) heap of ancillas $\xi \in \mathbf{AncHeap}$ called an *ancilla heap*, 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 return previously used ancillas if any are available, hence not using any extra space.

The function compile-BExp, shown in pseudo-code below, takes a Boolean expression B and a target bit i and then generates a reversible circuit computing $i \oplus B$. Note that ancillas are only allocated to store sub-expressions of $\wedge$ expressions, since $i \oplus (B_1 \oplus B_2) = (i \oplus B_1) \oplus B_2$ and so we compile $i \oplus (B_1 \oplus B_2)$ by first computing $i' = i \oplus B_1$, followed by $i' \oplus B_2$.

```
function compile-BExp($B$, $i$, $\xi$)
    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($B_1$, $i$, $\xi$)::compile-BExp($B_2$, $i$, $\xi$)
    else // $B = B_1 \wedge B_2$
        $a_1 \leftarrow$ pop-min($\xi$); $C \leftarrow$ compile-BExp($B_1$, $a_1$, $\xi$);
        $a_2 \leftarrow$ pop-min($\xi$); $C' \leftarrow$ compile-BExp($B_2$, $a_2$, $\xi$);
        $C$ :: $C'$ :: Toffoli $a_1$ $a_2$ $i$
    end if
end function
```

Cleanup The definition of compile-BExp above leaves many garbage bits that take up space and need to be cleaned before they can be re-used. To reclaim those bits, we clean temporary expressions after every call to compile-BExp.

To facilitate the cleanup – or *uncomputing* – of a circuit, we define the *restricted inverse* $\mathsf{uncompute}(C, A)$ 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:

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

The other cases are defined similarly. Note that since uncompute produces a subsequence of the original circuit C, no ancillary bits are used.

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 $C :: \mathsf{uncompute}(C, \{i\})$ maps a state s to $s[i \mapsto [\![B]\!]s \oplus s(i)]$, allowing any newly allocated ancillas to be pushed back onto the heap. Intuitively, since no bits contained in the set A are modified, the restricted inverse preserves their values; that the restricted inverse uncomputes the values of the remaining bits is less obvious, 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 5.

4.2 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 [21]) and 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 ReVerC as a *partial evaluator* parameterized by an abstract machine for evaluating Boolean expressions. As a side effect, we arrive at a unique model for circuit compilation similar to staged computation (see, e.g., [12]).

ReVerC works by evaluating the program with an abstract machine providing mechanisms for initializing and assigning locations on the store to Boolean expressions. We call an instantiation of this abstract machine an *interpretation* $\mathscr{I}$, which consists of a domain D equipped with two operators:

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

We typically denote an element of an interpretation domain D by σ. A sequence 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, an element of D 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 $\mathscr{I}_{standard}$ has domain $\mathbf{Store} = \mathbb{N} \rightharpoonup \mathbb{B}$, together with the operations

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

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

Example 3. The *reversible circuit* interpretation $\mathscr{I}_{circuit}$ has domain $D_{circuit} = (\mathbb{N} \rightharpoonup \mathbb{N}) \times \mathbf{Circ} \times \mathbf{AncHeap}$. In particular, given $(\rho, C, \xi) \in D_{circuit}$, ρ maps heap locations to bits in C, and ξ is an ancilla heap. Assignment and evaluation are further defined as follows:

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

Interpreting a program with $\mathscr{I}_{circuit}$ builds a reversible circuit executing the program, together with a mapping from heap locations to bits. Since the circuit is required to be reversible, 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 the bit associated with a heap location.

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

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

gives the operational semantics of Revs over the interpretation $\mathscr{I}$. We do not give a formal definition of $\Rightarrow_{\mathscr{I}}$, as it can be obtained trivially from the definition of $\Rightarrow$ (Figure 5) by replacing all heap updates with assign and taking $\mathsf{dom}(\sigma)$ to mean the set of locations on which eval is defined. To compile a program term t, ReVerC evaluates t over a particular interpretation $\mathscr{I}$ (for instance, the reversible circuit interpretation) and an initial heap $\sigma \in D$ according to the semantic relation $\Rightarrow_{\mathscr{I}}$. In this way, evaluating a program and compiling a program to a circuit look almost identical. This greatly simplifies the problem of verification (see Section 5).

ReVerC currently supports three modes of compilation, defined by giving interpretations: a default mode, an eagerly cleaned mode, and a "crush" mode. The default mode evaluates the program using the circuit interpretation, and simply returns the circuit and output bit(s), while the eager cleanup mode operates analogously, using instead the garbage-collected interpretation defined below in Section 4.3. The crush mode interprets a program as a list of Boolean expressions over free variables, which while unscalable allows highly optimized versions of small circuits to be compiled, a common practice in circuit synthesis. We omit the details of the Boolean expression interpretation.

Function compilation While the definition of ReVerC as a partial evaluator streamlines both development and verification, there is an inherent disconnect between the treatment of a (top-level) function expression by the interpreter and by the compiler, in that we want the compiler to evaluate the function body. Instead of defining a two-stage semantics for Revs we took the approach of applying a program transformation, whereby the function being compiled is evaluated on special heap locations representing the parameters. This creates a further problem in that the compiler needs to first determine the size of each parameter; to solve this problem, ReVerC performs a static analysis we call *parameter interference*. We omit the details of this analysis due to space constraints and instead point the interested reader to an extended version of this paper [3].

4.3 Eager cleanup

It was previously noted that the circuit interpretation allocates a new ancilla on every assignment to a location, due to the requirement of reversibility. Apart from ReVerC's additional optimization passes, this is effectively the Bennett method, and hence uses a large amount of extra space. One way to keep the space usage from continually expanding as assignments are made is to clean the old bit as soon as possible and then reuse it, rather than wait until the end of the computation. Here we develop an interpretation that performs this automatic, eager cleanup by augmenting the circuit interpretation with a *cleanup expression* for each bit. Our method is based on the eager cleanup of [21], and was intended as a more easily verifiable alternative to mutable dependency diagrams.

The *eager cleanup* interpretation $\mathscr{I}_{GC}$ has domain

$$D = (\mathbb{N} \rightharpoonup \mathbb{N}) \times \textbf{Circ} \times \textbf{AncHeap} \times (\mathbb{N} \rightharpoonup \textbf{BExp}),$$

where given $(\rho, C, \xi, \kappa) \in D$, ρ, C and ξ are as in the circuit interpretation. The partial function κ maps individual bits to a Boolean expression over the bits of C which can be used to return the bit to its initial state, called the cleanup expression. Specifically, we have the following property:

$$\forall i \in \mathsf{cod}(\rho), s'(i) \oplus [\![\kappa(i)]\!]s' = s(i) \qquad \text{where } s' = [\![C]\!]s.$$

Intuitively, any bit i can then be cleaned by simply computing $i \mapsto i \oplus \kappa(i)$, which in turn can be done by calling compile-BExp($\kappa(i)$, i).

Two problems remain, however. In general it may be the case that a bit *can not* be cleaned without affecting the value of other bits, as it might result in a loss of information – in the context of cleanup expressions, this occurs exactly when a bit's cleanup expression contains an irreducible self-reference. In particular, if $i \in \mathsf{vars}(B)$, then compile-BExp(B, i) does not compile a circuit computing $i \oplus B$ and hence won't clean the target bit correctly. In the case when a garbage bit contains a self-reference in its cleanup expression that can not be eliminated by Boolean simplification, ReVerC simply ignores the bit and performs a final round of cleanup at the end.

The second problem arises when a bit's cleanup expression references another bit that has itself since been cleaned or otherwise modified. In this case, the modification of the latter bit has invalidated the correctness property for the former bit. To ensure that the above relation always holds, whenever a bit is modified – corresponding to an XOR of the bit, i, with a Boolean expression B – all instances of bit i in every cleanup expression is replaced with $i \oplus B$.

Specifically we observe that, if $s'(i) = s(i) \oplus [\![B]\!]s$, then

$$s'(i) \oplus [\![B]\!]s = s(i) \oplus [\![B]\!]s \oplus [\![B]\!]s = s(i).$$

The function CLEAN, defined below, performs the cleanup of a bit i if possible, and validates all cleanup expressions in a given element of D:

```
function CLEAN($(\rho, C, \xi, \kappa)$, $i$)
    if $i \in \mathsf{vars}(\kappa(i))$ then return $(\rho, C, \xi, \kappa)$
    else
        $C' \leftarrow$ COMPILE-BEXP$(\kappa(i), i, \xi)$
        if $i$ is an ancilla then insert$(i, \xi)$
        $\kappa' \leftarrow \kappa[i' \in \mathsf{dom}(\kappa) \mapsto \kappa(i')[i \mapsto i \oplus \kappa(i)]]$
        return $(\rho, C :: C', \xi, \kappa')$
    end if
end function
```

Assignment and evaluation are defined in the eager cleanup interpretation as follows. Both are effectively the same as in the circuit interpretation, except the assignment operator calls CLEAN on the previous bit mapped to l.

$$\begin{aligned}
&\mathsf{assign}_{GC}((\rho, C, \xi, \kappa), l, B) = \textsc{clean}((\rho[l \mapsto i], C :: C', \xi, \kappa[i \mapsto B']), i)\\
&\quad \text{where } i = \text{pop-min}(\xi),\\
&\qquad B' = B[l' \in \mathsf{vars}(B) \mapsto \rho(l')]\\
&\qquad C' = \textsc{compile-BExp}(B', i, \xi)\\
&\mathsf{eval}_{GC}((\rho, C, \xi, \kappa), l, s)) = ([\![C]\!]s)\,(\rho(l))
\end{aligned}$$

The eager cleanup interpretation coincides with a reversible analogue of *garbage collection* for a very specific case when the number of references to a heap location (or in our case, a bit) is trivially zero. In fact, the CLEAN function can be used to eagerly clean bits that have no reference in other contexts. We intend to expand ReVerC to include a generic garbage collector that uses cleanup expressions to more aggressively reclaim space – for instance, when a bit's unique pre-image on the heap leaves the current scope.

4.4 Optimizations

During the course of compilation it is frequently the case that more ancillas are allocated than are actually needed, due to the program structure. For instance, when compiling the expression $i \leftarrow B$, if B can be factored as $i \oplus B'$ the assignment may be performed reversibly rather than allocating a new bit to store the value of B. Likewise if i is provably in the 0 or 1 state, the assignment may be performed reversibly without allocating a new bit. Our implementation identifies some of these common patterns, as well as general Boolean expression simplifications, to further minimize the space usage of compile circuits. All such optimizations in ReVerC have been formally verified.

5 Verification

In this section we describe the formal verification of ReVerC and give the major theorems proven. All theorems given in this section have been formally specified and proven using the F* compiler [27]. We first give theorems about our Boolean expression compiler, then use these to prove properties about whole program compilation. The total verification of the ReVerC core's approximately 2000 lines of code comprises around 2200 lines of F* code, and took just over 1 person-month. We feel that this relatively low-cost verification is a testament to the increasing ease with which formal verification can be carried out using modern proof assistants. Additionally, the verification relies on only 11 unproven axioms regarding simple properties of lookup tables and sets, such as the fact that a successful lookup is in the codomain of a lookup table.

Rather than give F* specifications, we translate our proofs to mathematical language as we believe this is more enlightening. The full source code of ReVerC including proofs can be obtained at https://github.com/msr-quarc/ReVerC.

5.1 Boolean expression compilation

Correctness Below is our main theorem establishing the correctness of the function 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, and if the ancilla bits are 0-valued, then the circuit computes the expression $i \oplus B$.

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

$$(\llbracket \textsc{compile-BExp}(B, i, \xi) \rrbracket s)\,(i) = s(i) \oplus \llbracket B \rrbracket s.$$

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 lemma 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.

Lemma 1. *Let C be a well-formed reversible circuit and $A \subset \mathbb{N}$ be some set of bits. If $A \cap \mathsf{control}(C) = \emptyset$ then for all states $s, s' = \llbracket C :: \mathsf{uncompute}(C, A) \rrbracket s$ and any $i \notin A$,*

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

Lemma 1 largely relies on the following important lemma stating in effect that the action of a circuit is determined by the values of the bits used as controls:

Lemma 2. *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 $\mathsf{control}(C) \subseteq A$, then*

$$([\![C]\!]s)(i) = ([\![C]\!]s')(i)$$

for all $i \in A$.

Lemma 1, together with the fact that COMPILE-BEXP produces a well-formed circuit under disjointness constraints, gives us our cleanup theorem below that Boolean expression compilation with cleanup correctly reverses the changes to every bit except the target.

Theorem 2. *Let B be a Boolean expression, ξ be an ancilla heap and $i \in \mathbb{N}$ such that $\mathsf{vars}(B)$, ξ and $\{i\}$ are all disjoint. Suppose* COMPILE-BEXP$(B, i, \xi) = C$*. Then for all $j \neq i$ and states s we have*

$$\left([\![C \circ \mathsf{uncompute}(C, \{i\})]\!]s\right)(j) = s(j).$$

5.2 REVS compilation

It was noted in Section 4 that the design of REVERC as a partial evaluator simplifies proving correctness. We expand on that point now, and in particular show that if a relation between elements 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 $\mathscr{I}_1, \mathscr{I}_2$ be interpretations and suppose whenever $(\sigma_1, \sigma_2) \in R$ for some relation $R \subseteq \mathscr{I}_1 \times \mathscr{I}_2$,*

$$(\mathsf{assign}_1(\sigma_1, l, B), \mathsf{assign}_2(\sigma_2, l, B)) \in R$$

for any l, B. Then for any term t, if $\langle t, \sigma_1 \rangle \Rightarrow_{\mathscr{I}_1} \langle v_1, \sigma_1' \rangle$ and $\langle t, \sigma_2 \rangle \Rightarrow_{\mathscr{I}_2} \langle v_2, \sigma_2' \rangle$, then $v_1 = v_2$ and $(\sigma_1', \sigma_2') \in R$.

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 we found this significantly reduces boilerplate proof code that is otherwise currently necessary in F* due to a lack of automated induction.

Given two interpretations $\mathscr{I}, \mathscr{I}'$, we say elements σ and σ' of $\mathscr{I}$ and $\mathscr{I}'$ are *observationally equivalent* with respect to a supplied set of initial values $s \in$ **State** if for all $i \in \mathbb{N}$, $\mathsf{eval}_{\mathscr{I}}(\sigma, i, s) = \mathsf{eval}_{\mathscr{I}'}(\sigma', i, s)$. We say $\sigma \sim_s \sigma'$ if σ and σ' are observationally equivalent with respect to s. As observational equivalence of two domain elements σ, σ' implies that any location in scope has the same valuation in either interpretation, it suffices to show that any compiled circuit is observationally equivalent to the standard interpretation. The following lemmas are used along with Theorem 3 to establish this fact for the default and eager-cleanup interpretations – a similar lemma is proven in the implementation of REVERC for the crush mode.

Lemma 3. *Let σ, σ' be elements of $\mathscr{I}_{standard}$ and $\mathscr{I}_{circuit}$, respectively. 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*

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

Moreover, the ancilla heap remains 0-filled.

We say that $(\rho, C, \xi) \in D_{circuit}$ is *valid* with respect to $s \in \mathbf{State}$ if and only if $s(i) = 0$ for all $i \in \xi$. For elements of D_{GC} the validity conditions are more involved, so we introduce a relation, $\mathcal{V} \subseteq D_{GC} \times \mathbf{State}$, defining the set of valid domain elements:

$$\begin{aligned}((\rho, C, \xi, \kappa), s) \in \mathcal{V} \iff \quad & \forall i \in \xi, s(i) = 0 \land \forall l, l' \in \mathsf{dom}(\rho), \rho(l) \neq \rho(l') \\ & \land \forall i \in \mathsf{cod}(\rho), [\![i \oplus \kappa(i)]\!]([\![C]\!] s) = s(i)\end{aligned}$$

Informally, $\mathcal{V}$ specifies that all bits on the heap have initial value 0, that ρ is a one-to-one mapping, and that for every active bit i, XORing i with $\kappa(i)$ returns the initial value of i – that is, $i \oplus \kappa(i)$ *cleans* i.

Lemma 4. *Let σ, σ' be elements of $\mathscr{I}_{standard}$ and $\mathscr{I}_{GC}$, respectively. For all $l \in \mathbb{N}, B \in \mathbf{BExp}, s \in \mathbf{State}$, if $\sigma \sim_s \sigma'$ and $(\sigma', s) \in \mathcal{V}$, then*

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

Moreover, $(\mathsf{assign}_{GC}(\sigma', l, B), s) \in \mathcal{V}$.

By setting the relation R_{GC} as

$$(\sigma_1, \sigma_2) \in R_{GC} \iff \sigma_2 \in \mathcal{V} \land \sigma_1 \sim_{s_0} \sigma_2$$

for $\sigma_1 \in D_{standard}$, by Theorem 3 and Lemma 4 it follows that partial evaluation/compilation preserves observational equivalence between $\mathscr{I}_{standard}$ and $\mathscr{I}_{GC}$. A similar result follows for $\mathscr{I}_{circuit}$.

To formally prove correctness of the compiler we need initial values in each interpretation (and an initial state) which are observationally equivalent. We don't describe the initial values here as they are dependent on the program transformation applied to expand top-level functions.

6 Experiments

We ran experiments to compare the bit, gate and Toffoli counts of circuits compiled by ReVerC to the original Revs compiler. The number of Toffoli gates in particular is distinguished as such 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* [19], or up to hundreds of times in most fault-tolerant architectures [2]. We compiled circuits for various arithmetic and cryptographic functions written in Revs using both compilers and reported the results in Table 1. Experiments were run in Linux using 8 GB of RAM.

The results show that both compilers are more-of-less evenly matched in terms of bit counts across both modes, despite ReVerC being certifiably correct.

Benchmark	Revs			Revs (eager)			ReVerC			ReVerC (eager)		
	bits	gates	Toffolis	bits	gates	Toffolis	bits	gates	Toffolis	bits	gates	Toffolis
carryRippleAdd 32	129	281	**62**	129	467	124	128	281	**62**	**113**	361	90
carryRippleAdd 64	257	569	**126**	257	947	252	256	569	**126**	**225**	745	186
mult 32	128	6016	4032	128	6016	4032	128	6016	4032	128	6016	4032
mult 64	256	24320	16256	256	24320	16256	256	24320	16256	256	24320	16256
carryLookahead 32	160	345	**103**	**109**	1036	344	165	499	120	146	576	146
carryLookahead 64	424	1026	**307**	**271**	3274	1130	432	1375	336	376	1649	428
modAdd 32	65	188	62	65	188	62	65	188	62	65	188	62
modAdd 64	129	380	126	129	380	126	129	380	126	129	380	126
cucarroAdder 32	65	98	32	65	98	32	65	98	32	65	98	32
cucarroAdder 64	129	194	64	129	194	64	129	194	64	129	194	64
ma4	17	24	8	17	24	8	17	24	8	17	24	8
SHA-2 round	449	1796	**594**	**353**	2276	754	452	1796	**594**	449	1796	**594**
MD5	7841	81664	**27520**	7905	82624	27968	4833	70912	**27520**	**4769**	70912	**27520**

Table 1. Bit and gate counts for both compilers in default and eager cleanup modes. In cases when not all results are the same, entries with the fewest bits used or Toffolis are bolded.

ReVerC's eager cleanup mode never used more bits than the default mode, as expected, and in half of the benchmarks reduced the number of bits. Moreover, in the cases of the carryRippleAdder and MD5 benchmarks, ReVerC's eager cleanup mode produced circuits with significantly fewer bits than either of Revs' modes. On the other hand, Revs saw dramatic decreases in bit numbers for carryLookahead and SHA-2 with its eager cleanup mode compared to ReVerC.

While the results show there is clearly room for optimization of gate counts, they appear consistent with other verified compilers (e.g., [13]) 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 [1].

7 Conclusion

We have described our verified compiler for the Revs language, ReVerC. 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 two interpretations implemented in ReVerC, the circuit interpretation and a garbage collected interpretation, the latter of which refines the former by applying eager cleanup.

While ReVerC 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. We intend to address these issues in the future, and to further improve upon ReVerC's space usage.

References

1. Using the GNU Compiler Collection. Free Software Foundation, Inc. (2016), `https://gcc.gnu.org/onlinedocs/gcc/`
2. Amy, M., Maslov, D., Mosca, M., Roetteler, M.: A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems 32(6), 818–830 (2013)
3. Amy, M., Roetteler, M., Svore, K.M.: Verified compilation of space-efficient reversible circuits. arXiv e-prints (2016), `https://arxiv.org/abs/1603.01635`
4. Anticoli, L., Piazza, C., Taglialegne, L., Zuliani, P.: Towards quantum programs verification: From Quipper circuits to QPMC. In: Proceedings of the 8th international Conference on Reversible Computation (RC'16). pp. 213–219 (2016)
5. Bennett, C.H.: Logical reversibility of computation. IBM Journal of Research and Development 17, 525–532 (1973)
6. Beringer, L., Stewart, G., Dockins, R., Appel, A.: Verified compilation for shared-memory C. In: Programming Languages and Systems, vol. 8410, pp. 107–127. Springer LNCS (2014)
7. Chlipala, A.: 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). pp. 93–106. ACM (2010)
8. Claessen, K.: Embedded Languages for Describing and Verifying Hardware. PhD thesis, Chalmers University of Technology and Göteborg University (2001)
9. Fournet, C., Swamy, N., Chen, J., Dagand, P.E., Strub, P.Y., Livshits, B.: Fully abstract compilation to javascript. In: Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13). pp. 371–384. ACM (2013)
10. Green, A.S., LeFanu Lumsdaine, P., Ross, N.J., Selinger, P., Valiron, B.: Quipper: a scalable quantum programming language. In: Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'13). ACM (2013)
11. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC'96). pp. 212–219. ACM (1996)
12. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1993)
13. Leroy, X.: Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06). pp. 42–54. ACM (2006)
14. Lin, C.C., Jha, N.K.: RMDDS: Reed-Muller decision diagram synthesis of reversible logic circuits. Journal on Emerging Technologies in Computing Systems 10(2), 14 (2014)
15. Markov, I.L.: Limits on fundamental limits to computation. Nature 512, 147–154 (2014)
16. Maslov, D., Miller, D.M., Dueck, G.W.: Techniques for the synthesis of reversible Toffoli networks. ACM Transactions on Design Automation of Electronic Systems 12(4), 42 (2007)

17. Miller, D.M., Maslov, D., Dueck, G.W.: A transformation based algorithm for reversible logic synthesis. In: Proceedings of the 40th Annual Design Automation Conference (DAC'03). pp. 318–323 (2003)
18. Neis, G., Hur, C.K., Kaiser, J.O., McLaughlin, C., Dreyer, D., Vafeiadis, V.: Pilsner: A compositionally verified compiler for a higher-order imperative language. In: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP'15). pp. 166–178. ACM (2015)
19. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press, Cambridge, UK (2000)
20. Ömer, B.: Quantum programming in QCL. Master's thesis, Technical University of Vienna (2000)
21. Parent, A., Roetteler, M., Svore, K.M.: Reversible circuit compilation with space constraints. arXiv e-prints (2015), `https://arxiv.org/abs/1510.00377`
22. Perconti, J., Ahmed, A.: Verifying an open compiler using multi-language semantics. In: ACM Transactions on Programming Languages and Systems, vol. 8410, pp. 128–148. Springer LNCS (2014)
23. Saeedi, M., Markov, I.L.: Synthesis and optimization of reversible circuits. ACM Computing Surveys 45(2), 21 (2013)
24. Scherer, A., Valiron, B., Mau, S.C., Alexander, S., van den Berg, E., Chapuran, T.E.: Resource analysis of the quantum linear system algorithm. arXiv e-prints (2015), `https://arxiv.org/abs/1505.06552`
25. Shafaei, A., Saeedi, M., Pedram, M.: Reversible logic synthesis of k-input, m-output lookup tables. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE'13). pp. 1235–1240 (2013)
26. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Journal on Computing 26(5), 1484–1509 (1997)
27. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in $F^{\star}$. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16). pp. 256–270. ACM (2016)
28. Thomsen, M.K.: A functional language for describing reversible logic. In: Proceedings of the 2012 Forum on Specification and Design Languages (FDL'12). pp. 135–142. IEEE (2012)
29. Wille, R., Drechsler, R.: Towards a Design Flow for Reversible Logic. Springer (2010)
30. Wille, R., Grosse, D., Miller, D., Drechsler, R.: Equivalence checking of reversible circuits. In: Proceedings of the 39th IEEE International Symposium on Multiple-Valued Logic (ISMVL'09). pp. 324–330 (2009)
31. Wille, R., Offermann, S., Drechsler, R.: Syrec: A programming language for synthesis of reversible circuits. In: Proceedings of the 2010 Forum on Specification and Design Languages (FDL'10). pp. 1–6 (2010)
32. Yamashita, S., Markov, I.: Fast equivalence-checking for quantum circuits. In: Proceedings of the 2010 IEEE/ACM Symposium on Nanoscale Architectures (NANOARCH'10). pp. 23–28 (2010)
33. Yokoyama, T., Glück, R.: A reversible programming language and its invertible self-interpreter. In: Proceedings of the 2007 Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'07). pp. 144–153. ACM (2007)