Ricercar: A Language for Describing and Rewriting Reversible Circuits with Ancillae and Its Permutation Semantics

Michael Kirkedal Thomsen[1](✉), Robin Kaarsgaard[2], and Mathias Soeken[1]

[1] Group of Computer Architecture, University of Bremen, Bremen, Germany
{kirkedal,msoeken}@informatik.uni-bremen.de
[2] DIKU, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark
robin@di.ku.dk

Abstract. Previously, Soeken and Thomsen presented six basic semantics-preserving rules for rewriting reversible logic circuits, defined using the well-known diagrammatic notation of Feynman. While this notation is both useful and intuitive for describing reversible circuits, its shortcomings in generality complicates the specification of more sophisticated and abstract rewriting rules.

In this paper, we introduce *Ricercar*, a general textual description language for reversible logic circuits designed explicitly to support rewriting. Taking the not gate and the identity gate as primitives, this language allows circuits to be constructed using control gates, sequential composition, and ancillae, through a notion of *ancilla scope*. We show how the above-mentioned rewriting rules are defined in this language, and extend the rewriting system with five additional rules to introduce and modify ancilla scope. This treatment of ancillae addresses the limitations of the original rewriting system in rewriting circuits with ancillae in the general case.

To set Ricercar on a theoretical foundation, we also define a permutation semantics over symmetric groups and show how the operations over permutations as transposition relate to the semantics of the language.

Keywords: Reversible logic · Term rewriting · Ancillae · Circuit equivalence · Permutation

1 Introduction

In [14] two of the authors presented six elementary rules for rewriting reversible circuits using mixed-polarity multiple-control Toffoli gates. Building on this,

M.K. Thomsen—This work was partly funded by the *European Commission* under the *7th Framework Programme.*

M.K. Thomsen—A preliminary version of Ricercar was presented as work-in-progress at *6th Conference on Reversible Computation, 2014.*

more complex rules, such as moving and deletion rules, can be derived. Rewriting using such rules can be used not just to reduce the size and cost of reversible circuits, but also to analyse and explain other optimisation approaches for reversible circuits. As one example, the templates presented in [12] are all derivable from these rewriting rules.

The rewriting rules in [14] are based on the diagrammatic notation first introduced by Feynman. This notation gives a very intuitive description of reversible circuits and the presented rewriting rules inherit this benefit. However, one goal with rewriting is to provide computer aid to the design of reversible circuits, and just as intuitive as diagrammatic notation is to understand for humans, just as hard it is to model for computers. In particular, representing the more general rules poses a problem.

In this paper we introduce *Ricercar*, a description language for reversible logic circuits (Sect. 3.) inspired by work on a *reversible combinator language* [18] and the *logic of reversible structures* [11]. Its only basic atoms are the not gate and the identity gate (both with named wires) from which other circuits are constructed using control gates and sequential composition. After describing the syntax and semantics of the language, we show how to define the graphical rewriting rules of [14] as textual rewriting rules for Ricercar descriptions (Sect. 4.) To give a theoretical foundation for Ricercar, we also define a permutation semantics over symmetric groups (Sect. 2) and show how the operations over permutations as transposition relate to the semantics of the language (Sect. 3.3).

A notable feature of the language is that it directly supports ancillae. Since reversible circuit logic does not support arbitrary fan-out, ancillae are often used to store partial results from computations by means of reversible duplication. The concept of ancillae have, however, been used in many different ways, but in this work we take the most strict possible definition. By ancillae we mean a variable (or a line) that are, for all possible assignments of other defined variables, guaranteed to be unchanged over the execution of a circuit.

This definition is much more strict than what is normally characterised by temporary storage, but it *is* needed if one wants to ensure that information is leaked and, thus, the backwards semantics of the circuits can be used directly. It is, however, still very useful in both high-level programs as well as reversible circuit constructs. As an example, an n-bit binary adder of linear depth can be implemented without ancillae, but it requires the use of reversible gates that have n inputs. However, using just one ancilla line the linear depth V-shaped adder [5,20] is implemented using only gates with a constant number of inputs. Furthermore, all current designs for implementing sub-linear depth adders require a larger number of ancilla lines that is dependent on the input size [7,17,19]. Using a similar definition, the *restore* model [4] has been investigated with respect to is computational complexity limits.

In Sect. 5 we discuss the *ancilla scope* construct of Ricercar and show five basic rewriting rules for inserting and modifying ancilla wires (Sect. 5.1). This is interesting given that deciding if a wire is indeed an ancilla wire is difficult; it can generally be done using equivalence checking, which for reversible circuits has been shown to be coNP-complete [10]. Furthermore, we show how to derive more general rules (Sect. 5.2), and show a non-trivial and useful example of how these can be used to create reversible circuits with ancilla wires from ancilla-free circuits (Sect. 5.3). As a result, the proposed rewriting language can serve as a framework to formally analyse the trade-off between gate count and number of ancilla lines in reversible circuits. Such trade-offs have so far been investigated theoretically for Turing machines in *e.g.* [2,3] and experimentally for reversible circuit synthesis in [21]. We discuss further related work in Sect. 6.

The main contributions of the paper are the following:

1. An extension of the rewriting rules with rules for circuit rewriting using ancillae.
2. A textual language to describe rewriting which is more concise than the diagrammatic notation.
3. Semantics for the rewrite rules based on permutations that is useful to show soundness of the rules and to formally argue over them.

2 Symmetric Groups as a Theory of Reversible Logic

Every reversible function f computed by a reversible circuit of n input lines $x_1, \dots, x_n$ and n output lines $y_1, \dots, y_n$ can be represented by an element π_f of the symmetric group S_{2^n}, *i.e.*, a permutation of the set $\{0, \dots, 2^n - 1\}$. We have $\pi_f(x) = y$ whenever $f(x_1, \dots, x_n) = (y_1, \dots, y_n)$, where x and y denote the natural number representations of the bits $x_1, \dots, x_n$ and $y_1, \dots, y_n$, respectively. This duality has been used for reversible logic synthesis in the last decade [6,13], but has also seen use as a theoretical foundation for the analysis of reversible circuit logic [15,16].

Unlike the usual formulation of the symmetric group S_n, we will consider its elements to be permutations of the set $\{0, \dots, n-1\}$ rather than $\{1, \dots, n\}$. However, we will use the standard notation of writing explicit permutations using square brackets, *e.g.* $\pi = [0\,1\,3\,2]$, cycles using parentheses, *e.g.* $\pi = (2, 3)$, and π_e for the identity permutation. Under this interpretation, composition of gates corresponds to multiplication (*i.e.*, composition) of permutations.

The gate library we consider consists of only single-target gates, which are characterised by changing one circuit line based on a control function that argues over the variables of the remaining lines. Since all single-target gates are self-inverse, their respective permutations are involutions with cycle representations consisting of only transpositions and fixpoints. As pointed out in [15], all such transpositions are of the form (a, b) where the hamming distance of a and b is 1, *i.e.*, their binary expansions differ in exactly one position. We refer to the set of all such transpositions as

$$H_n = \{(a, b) \mid \nu(a \oplus b) = 1\} \tag{1}$$

where ν denotes the sideways sum. Note that each transposition (a, b) in H_n corresponds to one fully controlled Toffoli gate with positive and negative control lines, acting on line i, where i is the single index for which $a_i \neq b_i$. The polarity of the controls is chosen according to the other bits. Based on this observation, we partition the set H_n into n sets $H_{n,1}, H_{n,2}, \dots, H_{n,n}$ such that

$$H_{n,i} = \{(a,b) \in H_n \mid a \oplus b = 2^{i-1}\} \tag{2}$$

contains all transpositions in which the components differ in their i-th bit. Single-target gates that act on the target line i are all permutations that consist of a subset of transpositions in $H_{n,i}$.

We call $g_n(f) \in S_{2^n}$ a *transposition generation function* which takes as argument an injective function $f : \{0, \dots, 2^n - 1\} \hookrightarrow \{0, \dots, 2^n - 1\}$ and returns the permutation

$$(0, f(0))(1, f(1)) \cdots (2^n - 1, f(2^n - 1)). \tag{3}$$

3 Ricercar: A Description Language for Reversible Logic

In this section, we will explain the description language, Ricercar, that is used to formulate the rewriting rules. We will first explain the syntax (Fig. 1) and then show two ways to describe the semantics.

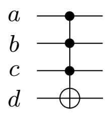
3.1 Syntax

Circuit wires (denoted by lower case Latin letters in the end of the alphabet: $\dots, x, y, z$) are defined over a set of names Σ that includes both input/output wires and ancilla wires currently in scope. (For wires without specific names, we will use lower case Latin letters starting from a.) We define a circuit (denoted by upper case Latin letters) to be one of the following five forms:

– The identity gate on a wire x, written $\mathsf{Id}(x)$, where $x \in \Sigma$.
– The not gate applied to a wire x, written $\mathsf{Not}(x)$, where $x \in \Sigma$.
– Sequential composition of two circuits, written using the operator " ; ".

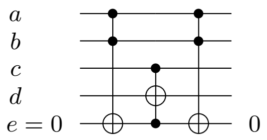
$A, B, C ::= \mathsf{Id}(x) \mid \mathsf{Not}(x)$	Identity and not gate
$\mid A \,;\, B$	Sequence of circuits
$\mid \phi \bullet\!\!- A$	Controlled circuit
$\mid \alpha x.A$	Scope of ancilla variable α; α is part of the syntax
$\phi, \psi, \pi ::= x \mid \neg\phi \mid \phi \wedge \phi$	Boolean formulas
$\mid \top \mid \bot \mid \phi \vee \phi \mid \phi \oplus \phi$	Derivable Boolean operators

Fig. 1. Syntax of Ricercar. Note that this grammar does not guarantee reversibility in itself. By x we mean that variables occurring in Boolean formulas must be elements from a predefined set of input/output wires or ancilla wires in scope.



$(a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d)$

(a) Three-controlled Toffoli gate



$\alpha e.((a \wedge b) \bullet\!\!- \mathsf{Not}(e)\ ;\ (c \wedge e) \bullet\!\!- \mathsf{Not}(d)\ ;\ (a \wedge b) \bullet\!\!- \mathsf{Not}(e))$

(b) Sequence of Toffoli gates

Fig. 2. Two (weakly) equivalent reversible circuits and their descriptions in Ricercar. Here, e denotes an arbitrary ancilla wire.

$$\mathrm{inv}(\mathsf{Id}(x)) = \mathsf{Id}(x) \qquad \mathrm{inv}(A\ ;\ B) = \mathrm{inv}(B)\ ;\ \mathrm{inv}(A) \qquad \mathrm{inv}(\alpha x.A) = \alpha x.\,\mathrm{inv}(A)$$
$$\mathrm{inv}(\mathsf{Not}(x)) = \mathsf{Not}(x) \qquad \mathrm{inv}(\phi \bullet\!\!- A) = \phi \bullet\!\!- \mathrm{inv}(A)$$

Fig. 3. The syntactic function inv(·) that defines the inverse of a Ricercar description

– A controlled circuit, denoted with the binary "$\bullet\!\!-$" operator, which contains a control function ϕ and a controlled circuit A.[1] The control function can be any Boolean formula.
– An ancilla scope for a circuit A, denoted with a functional lambda-style notation using the symbol α, and a variable denoting a wire which *must* be false both before and after A. Without loss of generality, we will assume that ancillae scopes always introduces fresh variable names.

For readability, we define control gates ($\bullet\!\!-$) to bind tighter than sequence (;) and the unary ($\alpha x.$).

Figure 2 shows two example circuits, defined using multiply controlled Toffoli gates, represented in the usual diagram notation due to Feynman, as well as in Ricercar.

As Ricercar should be reversible, we will define the straight-forward inverse of all the syntactic constructs. We have chosen *not* to include inversion as a basic construct, but will define it as a syntactic function; this simplifies both the language and the following rewriting rules. Figure 3 shows the inversion function inv(·).

3.2 Ancillae and Reversibly Well-Formed Properties

Ancillae hold a central place in Ricercar. We follow the idea that there are always as many ancilla wires available as needed. Consequently ancilla lines do not need to be declared in advance, but can be introduced on-the-fly. This is not an unrealistic assumption: remember that we define ancillae to be constant (false) at both input and output, which permits a large degree of reuse. Furthermore,

[1] The "$\bullet\!\!-$" notation is borrowed from [11], although with a different semantics.

$$\boxed{\begin{array}{ccc} \mathrm{rwf}(\mathsf{Id}(x)) = \{x\} & \mathrm{rwf}(A\ ;\ B) = \mathrm{rwf}(A) \cup \mathrm{rwf}(B) & \\ & & \mathrm{rwf}(\alpha x.A) = \mathrm{rwf}(A) \backslash \{x\} \\ \mathrm{rwf}(\mathsf{Not}(x)) = \{x\} & \mathrm{rwf}(\phi \bullet\!\!- A) = \mathrm{dom}(\phi) \uplus \mathrm{rwf}(A) & \end{array}}$$

Fig. 4. A reversible circuit, A, is reversibly well-formed iff $\mathrm{rwf}(A)$ evaluates to a set. Here, $\uplus$ is the disjoint union and $\mathrm{dom}(\phi)$ is the domain of the Boolean function ϕ. We assume that the disjoint union is undefined whenever the operands are not disjoint.

the actual number of ancilla lines used is limited by the depth of the circuit, and cannot grow unboundedly. As an example, given that we know the upper bound of the depth to implement a reversible circuit without ancillae (*cf.* [1]), this also gives an upper bound on the number of (useful) ancilla any circuit with smaller depth can have.

The syntax presented in Fig. 1 does not guarantee reversibility by itself. One problem comes from the control gate, where we must enforce that the wires of the control function are disjoint from wires of the circuit being controlled. This is similar to the concept used in the reversible updates in Janus [22]. Figure 4 shows a function $\mathrm{rwf}(\cdot)$ that implements this check; we say that the circuit is *reversibly well-formed* if it upholds this restriction. Given a circuit description A, it returns the set of all used variable names *if and only if* A is reversibly well-formed. If a circuit A is *not* reversibly well-formed, the disjoint union operation will fail on the control gate operator, and the result of $\mathrm{rwf}(A)$ will thus be undefined.

However, even a reversibly well-formed circuit is not necessarily reversible.[2] To ensure that an ancilla variable within an ancilla scope does indeed have ancilla behaviour (guaranteed false at both input and output), we need a additional semantic check. However, a circuit *without* ancillae is reversibly well-formed if and only if it is reversible. In Sect. 5, we will show how this can be exploited to introduce ancillae in a way that guarentees reversibility.

3.3 Operational Semantics

The straightforward semantics of Ricercar is shown in Fig. 5; they follow, but also extend, the logic by Fredkin and Toffoli, and describe the mapping from a circuit description to a reversible circuit using the well-known gates. More concretely, this semantics can be used to show that Ricercar is actually reversible.

Theorem 1 (Reversibility). *For all mappings σ and circuits A there exists a mapping σ' and a circuit B such that*

$$\sigma \vdash A \to \sigma' \iff \sigma' \vdash B \to \sigma.$$

This theorem and the following two lemmas are easily proven by structural induction over the circuit A and reference to the operational semantics of Ricercar (Fig. 5).

[2] The other direction holds: all reversible circuits are reversibly well-formed.

$$\frac{\sigma : \Sigma \rightharpoonup \mathbb{B} \qquad \sigma \vdash \neg x \to b}{\sigma \vdash \mathsf{Not}(x) \to \sigma[x \mapsto b]} \qquad \frac{}{\sigma \vdash \mathsf{Id}(x) \to \sigma}$$

$$\frac{\sigma \vdash A \to \sigma'' \qquad \sigma'' \vdash B \to \sigma'}{\sigma \vdash A \ ; \ B \to \sigma'}$$

$$\frac{\sigma \vdash \phi \to 1 \qquad \sigma \vdash A \to \sigma'}{\sigma \vdash \phi \bullet\!\!- A \to \sigma'} \qquad \frac{\sigma \vdash \phi \to 0}{\sigma \vdash \phi \bullet\!\!- A \to \sigma}$$

$$\frac{\sigma \vdash x \to b \qquad \sigma[x \mapsto 0] \vdash A \to \sigma' \qquad \sigma' \vdash x \to 0}{\sigma \vdash \alpha x.A \to \sigma'[x \mapsto b]}$$

Fig. 5. The semantics of Ricercar. Here, σ is a partial function mapping variable names to Boolean values; any variable name that is not part of the input is assumed to be undefined in σ. The semantics uses two judgment forms, $\sigma \vdash A \to \sigma'$ for evaluating circuits, and $\sigma \vdash \phi \to b$ for evaluating Boolean formulae, both with respect to σ. The rules for judgments of the latter form are not shown, but are completely standard.

To ensure that the previously defined inversion (with sequence as composition function) is indeed inversion, we show the following.

Lemma 1 (Inversion). *For all circuits A and states σ,*

$$\sigma \vdash A \ ; \ \mathrm{inv}(A) \to \sigma \quad \textit{and} \quad \sigma \vdash \mathrm{inv}(A) \ ; \ A \to \sigma.$$

Later it will also be useful to know that the inversion function respects involution symmetry.

Lemma 2 (Involution Symmetry). *For all circuits A, and states σ and σ',*

$$\sigma \vdash A \to \sigma' \iff \sigma \vdash \mathrm{inv}(\mathrm{inv}(A)) \to \sigma'.$$

3.4 Permutation (Denotational) Semantics

In order to ease the formal analyses using this language, we also express the functional semantics in terms of permutations. The counterparts to Id, Not, and '$\bullet\!\!-$' are provided for this purpose. In contrast to the language, the permutation description requires an order of variables and therefore we assume a strict total order '>' on the variables in Σ for the following equations. If $x > y$, it means that the variable x corresponds to a more significant bit than y. For the identity and the not gate we have

$$\mathsf{Id}(x) = \pi_e \quad \text{and} \quad \mathsf{Not}(x) = (0,1) \qquad \text{if } \Sigma = \{x\}. \tag{4}$$

For the following four equations, let $G(\pi, f)$ be the commutator $g_n(f) \circ \pi \circ g_n^{-1}(f)$ for a permutation $\pi \in S_{2^n}$ and a function f as in (3). Note that G is an endomorphism with respect to composition, since

$$\begin{aligned} G(\pi_1 \circ \pi_2, f) &= g_n(f) \circ \pi_1 \circ \pi_2 \circ g_n^{-1}(f) \\ &= g_n(f) \circ \pi_1 \circ g_n^{-1}(f) \circ g_n(f) \circ \pi_2 \circ g_n^{-1}(f) \\ &= G(\pi_1, f) \circ G(\pi_2, f). \end{aligned}$$

For some circuit A, let π_A be its permutation representation. Then one can "add a control line from the bottom," expressed as

$$\neg x \mathrel{\bullet\!\!-} A = G(\pi_A, x \mapsto x) \quad \begin{array}{l} \text{if } \Sigma = \mathrm{rwf}(A) \cup \{x\} \\ \text{and } x > y \text{ for all } y \in \mathrm{rwf}(A) \end{array} \tag{5}$$

and

$$x \mathrel{\bullet\!\!-} A = G(\pi_A, x \mapsto x + 2^n) \quad \begin{array}{l} \text{with } n = |\,\mathrm{rwf}(A)|, \text{ if } \Sigma = \mathrm{rwf}(A) \cup \{x\} \\ \text{and } x > y \text{ for all } y \in \mathrm{rwf}(A). \end{array} \tag{6}$$

Similarly, one can "add a control line from the top," expressed as

$$\neg x \mathrel{\bullet\!\!-} A = G(\pi_A, x \mapsto 2x) \quad \begin{array}{l} \text{if } \Sigma = \mathrm{rwf}(A) \cup \{x\} \\ \text{and } x < y \text{ for all } y \in \mathrm{rwf}(A) \end{array} \tag{7}$$

and

$$x \mathrel{\bullet\!\!-} A = G(\pi_A, x \mapsto 2x + 1) \quad \begin{array}{l} \text{if } \Sigma = \mathrm{rwf}(A) \cup \{x\} \\ \text{and } x < y \text{ for all } y \in \mathrm{rwf}(A). \end{array} \tag{8}$$

The above denotational semantics is not complete. Circuit sequence (;) can be defined by permutation composition after extending the two permutations to the same symmetric group, and scoped ancillae can be accommodated by imposing restrictions on the permutation for the more general circuit (*i.e.*, where the ancilla is considered as any other input line.) It is then not hard to prove equivalence between the operational and denotational semantics. The denotational semantics is reversible by construction.

4 Rewriting in Ricercar

In this section, we will recap the rewriting rules from [14], and define the rules with respect to Ricercar, as well as show soundness based on the permutation semantics.

First, however, note that gate composition is associative; that is, in a cascade of gates, the order in which we look at the gates does not matter, so in, *e.g.* Fig. 2(b), we are free to either look at the two first gates and perform rewriting on these, or start with the last two gates instead. The identity gate is the identity element for sequences:

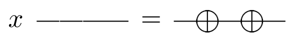
$$A = \mathsf{Id}(x) \; ; \; A = A \; ; \; \mathsf{Id}(x) \tag{ID}$$

Furthermore, note that we can always rewrite the controlling Boolean functions and, *e.g.* use identities from AND-EXOR decomposition:

$$\begin{aligned} \phi \bullet\!\!- \psi \bullet\!\!- A &= (\phi \wedge \psi) \bullet\!\!- A && \text{and} \\ \phi \bullet\!\!- A \,;\, \psi \bullet\!\!- A &= (\phi \oplus \psi) \bullet\!\!- A && \text{if } A = \mathrm{inv}(A). \end{aligned}$$

Finally, implicit to rules is that the circuits must always be reversibly well-formed both before and after a rewriting, and that in any given circuit we can rewrite any sub-circuit we like.

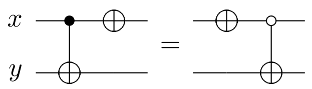
The first rule presented in [14] is for introducing and eliminating not gates, and states that we can always rewrite the identity function to two not gates.



$$\mathsf{Id}(x) = \mathsf{Not}(x) \,;\, \mathsf{Not}(x) \tag{R1}$$

Soundness trivially follows from $\pi_e = (0,1) \circ (0,1)$.

The second rule states that we can "move" a not gate over a control if we negate the control line.



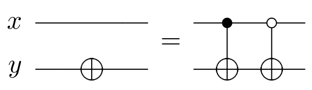
$$x \bullet\!\!- \mathsf{Not}(y) \,;\, \mathsf{Not}(x) = \mathsf{Not}(x) \,;\, \neg x \bullet\!\!- \mathsf{Not}(y) \tag{R2}$$

Similar to [14], we notice that its dual rule with negative control can be derived using this rule in combination with Rule R1:

$$\begin{aligned} \neg x \bullet\!\!- \mathsf{Not}(y) \,;\, \mathsf{Not}(x) &\overset{(R1)}{=} \mathsf{Not}(x) \,;\, \mathsf{Not}(x) \,;\, \neg x \bullet\!\!- \mathsf{Not}(y) \,;\, \mathsf{Not}(x) \\ &\overset{(R2)}{=} \mathsf{Not}(x) \,;\, x \bullet\!\!- \mathsf{Not}(y) \,;\, \mathsf{Not}(x) \,;\, \mathsf{Not}(x) \\ &\overset{(R1)}{=} \mathsf{Not}(x) \,;\, x \bullet\!\!- \mathsf{Not}(y). \end{aligned} \tag{R2'}$$

Soundness follows from Eqs. (5)–(8) and the identity $(a,b)(b,c) = (a,c)(a,b) = (a,c)(b,c)$.

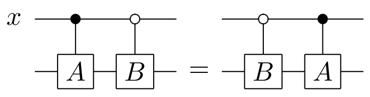
Third, we can extend a gate by copying it and adding once a positive and once a negative control line to it.



$$\mathsf{Id}(x) \,;\, \mathsf{Not}(y) = x \bullet\!\!- \mathsf{Not}(y) \,;\, \neg x \bullet\!\!- \mathsf{Not}(y) \tag{R3}$$

Soundness follows from Eqs. (7) and (8). In fact, in permutation notation, both controlled not gates are represented by a single transposition, and combining them results in the (permutation corresponding to the) not gate. Also, combining the equation of adding a negative and positive control yields an equation for adding an empty line.

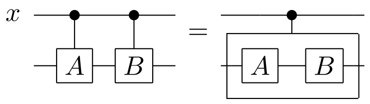
Next, two arbitrary adjacent gates can be interchanged whenever they have a common control line with different polarities. Notice how Ricercar captures the fact that two controlled circuits can have *any* circuit structure; something that is not well captured by the diagrammatic notation in [14].



$$x \bullet\!\!- A \;;\; \neg x \bullet\!\!- B = \neg x \bullet\!\!- B \;;\; x \bullet\!\!- A \qquad \text{(R4)}$$

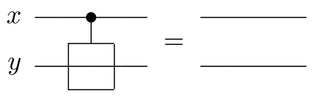
The permutation equations also reveal this property, since the transpositions resulting from $\neg x \bullet\!\!- A$ and $x \bullet\!\!- A$ are disjoint.

Whenever two gates share the same control variable with the same polarity, these two gates can be grouped together, where the group is controlled by that control line. Again, Ricercar allows for a precise formulation of the idea, compared to the diagrammatic notation.



$$x \bullet\!\!- A \;;\; x \bullet\!\!- B = x \bullet\!\!- (A \;;\; B) \qquad \text{(R5)}$$

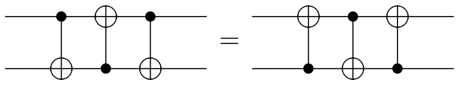
This property follows from G being an endomorphism. Finally, we have the rule for introducing and eliminating groups of wires.



$$x \bullet\!\!- \mathsf{Id}(y) = \mathsf{Id}(x) \;;\; \mathsf{Id}(y) \qquad \text{(R6)}$$

4.1 A Note on Completeness

A question raised in [14] regards the *completeness* of the above rules, in the sense that every circuit can be rewritten, in a finite number of steps, to any other equivalent circuit. In this strict sense, the rules are *not* complete. The counter example is the two-line swap gate, which can be represented in the following two ways:



Given the six rules, it is not possible to rewrite one to the other. This is, of course, not satisfactory, and a shortcoming that must be solved. The easy solution would be to add the above equation as a seventh rule, but the extent to which there exist other counter examples related to this problem is unknown, and the solution is only an incremental extension that will not add any interesting new insights.

However, this counter example is restricted in that it does not generalise to more lines. If we have a third line available (no matter its value), it can be used as an auxiliary line and thereby enable rewriting between the two swap gates. The question is now if the six rules are complete for circuits of more than two lines. But there is a possibility that two similar three line circuits exist and we need to assume a fourth auxiliary line to rewrite between them. The better solution, that we will follow in the next section, is thus to extend with rules for ancilla lines.

5 Ancillae and Rewriting

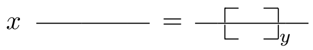
As mentioned earlier, ancillae is a powerful extension to a reversible language, but the power comes at a cost. Checking that some defined ancillae are indeed unchanged for all possible input vectors of an arbitrary description is hard; in general, one has to test all possible input vectors, which is undesirable. For a reversible programming language such as Janus [22], this is therefore implemented as a runtime check that checks the reversibility of a program *only* in relation to the executing input vector. This is also the case for the syntax described in Fig. 1.

For this reason, we will pursue a different approach. Given a description without ancillae, we can statically check reversibility using the rwf-function shown in Fig. 4. From a reversible description without ancillae, we will now define rewriting rules that can extend the given description with ancilla wires. Hence, instead of showing reversibility of a description with ancillae (which is hard), we only have to show that the rewriting rules do not interfere with the ancilla-property of the wires, and thereby with the reversibility of the circuit; this is much easier.

5.1 The Rewriting Rules

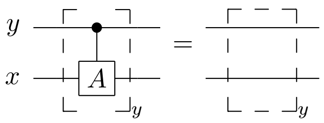
To be able to introduce and remove ancilla wires from a circuit, we have identified the need for five basic rules.

The first rule is for introducing and removing an ancilla scope. It states that we can always introduce a scope containing the identity circuit with a fresh (unused) ancilla wire name.



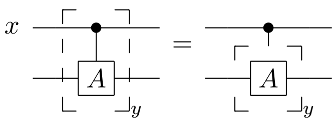
$$\mathsf{Id}(x) = \alpha y.\mathsf{Id}(x) \tag{A1}$$

The second rule states that a circuit in an ancilla scope can be removed (or added) if it is controlled by the ancilla wire. Recall that the ancilla variable is assumed to be assigned false outside of the ancilla scope, so the control is never active. For now, the gate must be the only gate within the scope, but we will show how this can be generalised later.



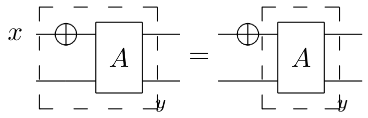
$$\alpha y.(y \bullet\!\!- A) = \alpha y.\mathsf{Id}(x), \qquad x \in \mathrm{rwf}(A) \tag{A2}$$

The third rule considers the case in which the controlling wire is not the ancilla wire of the scope. In this case, the control can be pulled out of the ancilla scope, and thereby control the scope containing the controlled circuit.



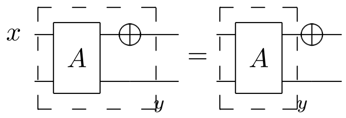
$$\alpha y.(x \bullet\!\!- A) = x \bullet\!\!- \alpha y.A, \qquad x \neq y \tag{A3}$$

The fourth rules states that a not gate on a non-ancilla wire that is positioned to the immediate left of a circuit it shares a scope with can be pulled out of the ancilla scope.



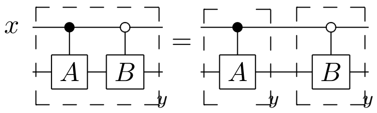
$$\alpha y.(\mathsf{Not}(x) \ ; \ A) = \mathsf{Not}(x) \ ; \ \alpha y.A, \qquad x \neq y \qquad \text{(A4)}$$

In the case where the not gate is on the right, a similar rule can be derived from the Involution Symmetry Lemma with Rule A4.



$$\alpha y.(A \ ; \ \mathsf{Not}(x)) = (\alpha y.A) \ ; \ \mathsf{Not}(x), \qquad x \neq y \qquad \text{(A4')}$$

The fifth and final rule states that if (and only if) an ancilla scope contains a sequence of two circuits where the first is positively controlled, and the second is negatively controlled by the same wire, then this scope can be divided into two; or, in the other direction, merged. Note that x *can* be equal to y. This rule is likely the most powerful of the five, and it shows up in the proofs that extend and generalise the previous rules.



$$\alpha y.(x \bullet\!\!- A \ ; \ \neg x \bullet\!\!- B) = (\alpha y.x \bullet\!\!- A) \ ; \ (\alpha y.\neg x \bullet\!\!- B) \qquad \text{(A5)}$$

That the first four rules (A1 to A4) do not interfere with the ancilla-property of a wire is clear, but the last rule (A5) requires an argument. Only either A or B (but not both) is performed as the control on x is exclusive. Thus assuming that $x \neq y$, any usage of y in A must have uncomputed y to zero again; similarly any usage of y in B must have assumed it to be zero. Therefore, we can divide the ancilla scope of y. If $x = y$ then y will always be unchanged (zero) as y is not used in B.

5.2 Generalisation of Ancilla Rules

Rule A2 has a twin-rule for the case where the gate is negatively controlled by the ancilla wire. We can derive that this is equal to the controlled gate in the following way:

$$\begin{aligned}\alpha y.(\neg y \bullet\!\!- A) &\overset{\text{(ID)}}{=} \mathsf{Id}(x) \ ; \ (\alpha y.\neg y \bullet\!\!- A) &&\overset{\text{(A1)}}{=} (\alpha y.\mathsf{Id}(x)) \ ; \ (\alpha y.\neg y \bullet\!\!- A) \\ &\overset{\text{(A2)}}{=} (\alpha y.y \bullet\!\!- A) \ ; \ (\alpha y.\neg y \bullet\!\!- A) &&\overset{\text{(A5)}}{=} \alpha y.(y \bullet\!\!- A \ ; \ \neg y \bullet\!\!- A) \\ &\overset{\text{(R3)}}{=} \alpha y.A. && \qquad \text{(A2')}\end{aligned}$$

This twin-rule can now be used to show the more general rule that if an ancilla wire controls a circuit in the beginning of a scope it can be removed entirely:

$$\begin{aligned}
\alpha y.(y \bullet\!\!- A \;;\; B) &\overset{(R3)}{=} \alpha y.(y \bullet\!\!- A \;;\; y \bullet\!\!- B \;;\; \neg y \bullet\!\!- B)\\
&\overset{(R5)}{=} \alpha y.(y \bullet\!\!- (A \;;\; B) \;;\; \neg\alpha \bullet\!\!- .B)\\
&\overset{(A5)}{=} (\alpha y.y \bullet\!\!- (A \;;\; B)) \;;\; (\alpha y.\neg y \bullet\!\!- B)\\
&\overset{(A2)}{=} (\alpha y.\mathsf{Id}(x)) \;;\; (\alpha y.\neg y \bullet\!\!- B) \quad \overset{(A2')}{=} \quad (\alpha y.\mathsf{Id}(x)) \;;\; (\alpha y.B)\\
&\overset{(A1)}{=} \mathsf{Id}(x) \;;\; \alpha y.B \qquad\qquad \overset{(ID)}{=} \quad \alpha y.B.
\end{aligned} \tag{D8}$$

Similarly, we can also generalise A3 to the case where the circuit in the ancilla scope contains more than one gate. Assuming that $x \neq y$, to extract x from the ancilla scope of y we can do

$$\begin{aligned}
\alpha y.x \bullet\!\!- A \;;\; B &\overset{(R3)}{=} \alpha y.(x \bullet\!\!- A \;;\; x \bullet\!\!- B \;;\; \neg x \bullet\!\!- B)\\
&\overset{(R5)}{=} \alpha y.(x \bullet\!\!- (A \;;\; B) \;;\; \neg x \bullet\!\!- B)\\
&\overset{(A5)}{=} (\alpha y.x \bullet\!\!- (A \;;\; B)) \;;\; (\alpha y.\neg x \bullet\!\!- B)\\
&\overset{(A3)}{=} x \bullet\!\!- (\alpha y.(A \;;\; B)) \;;\; \neg x \bullet\!\!- \alpha y.B.
\end{aligned} \tag{D9}$$

This duplicates B such that it is performed both when x is true, and when it is false. Assuming that the ancilla wire y does not occur in A (*i.e.* $y \notin \mathrm{rwf}(A)$), we can then use D9 to show by induction on the depth of the control that

$$\alpha y.(A \;;\; B) = A \;;\; \alpha y.B, \qquad y \notin \mathrm{rwf}(A). \tag{D10}$$

As a special case of this rule, specifically when $B = \mathsf{Id}(x)$ for any choice of $x \in \mathrm{rwf}(A) \cup \{y\}$, we get that

$$\alpha y.A = A, \qquad y \notin \mathrm{rwf}(A). \tag{D11}$$

As a closing derived rule, we will show how ancilla wires can be introduced to perform computations that were otherwise performed by an input wire. In other words, we can use the rules to introduce ancilla wires that are then used to control what was previously controlled by x.

$$\begin{aligned}
x \bullet\!\!- A &\overset{(D11)}{=} \alpha y.(x \bullet\!\!- A \;;\; \mathsf{Id}(y))\\
&\overset{(D1)}{=} \alpha y.(x \bullet\!\!- A \;;\; x \bullet\!\!- y \;;\; x \bullet\!\!- y)\\
&\overset{(D8)}{=} \alpha y.(y \bullet\!\!- A \;;\; x \bullet\!\!- A \;;\; x \bullet\!\!- y \;;\; x \bullet\!\!- y)\\
&\overset{(D6)}{=} \alpha y.(x \bullet\!\!- A \;;\; y \bullet\!\!- A \;;\; x \bullet\!\!- y \;;\; x \bullet\!\!- y)\\
&\overset{(D7)}{=} \alpha y.(x \bullet\!\!- y \;;\; y \bullet\!\!- A \;;\; x \bullet\!\!- y).
\end{aligned} \tag{D12}$$

Here D1, D6, and D7 refer to derived rules from [14]. This example increases the size and depth of the circuit, but if x controls several gates this can be used to reduce the depth of the circuit considering that gates can be put in parallel.

5.3 Practical Example of Application of Ricercar

As a final example we show how to derive the circuit depicted in Fig. 2(b) from the one in Fig. 2(a) using the rewriting rules. Again D1 and D7 refer to derived rules from [14].

$$
\begin{aligned}
&(a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d) \\
&\quad \overset{(\mathrm{D11})}{=} \alpha e.((a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d)) \\
&\quad \overset{(\mathrm{D8})}{=} \alpha e.((c \wedge \beta) \bullet\!\!- \mathsf{Not}(d) \ ; \ (a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d)) \\
&\quad \overset{(\mathrm{D1})}{=} \alpha e.((a \wedge b) \bullet\!\!- \mathsf{Not}(e) \ ; \ (a \wedge b) \bullet\!\!- \mathsf{Not}(e) \ ; \ (c \wedge e) \bullet\!\!- \mathsf{Not}(d) \ ; \\
&\qquad\quad (a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d)) \\
&\quad \overset{(\mathrm{D7})}{=} \alpha e.((a \wedge b) \bullet\!\!- \mathsf{Not}(e) \ ; \ (c \wedge e) \bullet\!\!- \mathsf{Not}(d) \ ; \ (a \wedge b) \bullet\!\!- \mathsf{Not}(e) \ ; \\
&\qquad\quad (a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d) \ ; \ (a \wedge b \wedge c) \bullet\!\!- \mathsf{Not}(d)) \\
&\quad \overset{(\mathrm{D1})}{=} \alpha e.((a \wedge b) \bullet\!\!- \mathsf{Not}(e) \ ; \ (c \wedge e) \bullet\!\!- \mathsf{Not}(d) \ ; \ (a \wedge b) \bullet\!\!- \mathsf{Not}(e)).
\end{aligned}
$$

6 Related Work

This is not the first language that has been designed to describe the concepts of reversible logic; there exist description languages for both reversible and quantum circuits.

The closest related work is the *Reversible Combinator Language* (RCL) [18] that was also made to describe reversible logic; though it is more general than our work, there are still some common ideas. Taking inspiration from RCL, we use a similar sequence operator, and the conditional in RCL is (in its semantics) comparable to our control operator. However, being a combinator language, RCL does not have variables, but rather a type system in which circuits of arbitrary size with a given structure can be defined. Also it has more general combinators, such as a ripple circuit and parallel composition, as basic constructs. RCL also admits a number of rewriting rules, but compared to Ricercar, RCL's type system and larger set of atomic gates makes rewriting more cumbersome.

Although aiming to describe quantum circuits, it is worth mentioning *Quipper* [8,9]. Though Quipper also supports ancilla scopes, in order to uphold the ancilla-property, the Quipper synthesis results in a symmetric compute-use-uncompute “Bennett-style” structure of the ancilla wires. In contrast, the ancilla scopes in Ricercar are more general, but have to be built from the bottom up with rewriting to uphold the property. The interested reader can find further references for quantum description languages in the works above.

7 Conclusion

In this paper we have presented *Ricercar*, a language designed to describe reversible circuits. A main focus during the design process of the language has been rewriting, specifically that rewriting rules should be easy to both define and apply in the language. The previous approach to rewriting of reversible circuits was shown for the standard diagrammatic notation, but this notation neither captures the full intent of all of the six original rules, nor does it provide an optimal setting for a future computer aided system. Ricercar, with its simple symbolic description, both captures the complete intent of the original rules, and has a syntax that is directly implementable.

In addition, Ricercar has support for ancillae as a basic circuit construct in the form of a *scope*. Using this construct, we have extended the six original rules with five basic rules that applies when rewriting ancillae. We have shown how it is possible to use these rules to derive more general ones that also apply to ancillae, and as a final example, how to derive a rule that moves the control of a gate from an input wire to an ancilla wire.

Determining reversibility of a circuit that contains ancillae is generally hard, but with the presented rewriting rules, it is possible to take an ancillae-free circuit (for which it is easy to show reversibility) and rewrite it into a circuit that contains ancillae, and is guaranteed to be reversible. The key here is that the basic rules (and all of the derived rules) cannot break the ancilla-property of a wire and, thus, the reversibility of the circuit.

We hope that this approach can further help in the understanding of the trade-off between ancillae on the one hand, and the size and depth of a reversible circuit on the other.

Acknowledgments. The authors thank the anonymous reviewers from RC'14 for their useful comments on a preliminary version of this paper, presented as work-in-progress at *6th Conference on Reversible Computation, 2014* and the anonymous reviewers from RC'15 on the current version. Finally, we thank Holger Bock Axelsen for several discussion on work in this paper.

References

1. Abdessaied, N., Soeken, M., Thomsen, M.K., Drechsler, R.: Upper bounds for reversible circuits based on Young subgroups. Information Processing Letters **114**(6), 282–286 (2014)
2. Bennett, C.H.: Time/Space Trade-Offs for reversible computation. SIAM Journal on Computing **18**(4), 766–776 (1989)
3. Buhrman, H., Tromp, J., Vitányi, P.: Time and space bounds for reversible simulation. Journal of Physics A: Mathematical and General **34**(35), 6821–6830 (2001)
4. Chan, T., Munro, J.I., Raman, V.: Selection and sorting in the "restore" model. In: Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 995–1004 (2014)
5. Cuccaro, S.A., Draper, T.G., Kutin, S.A., Moulton, D.P.: A new quantum ripple-carry addition circuit. arXiv:quant-ph/0410184v1 (2005)
6. De Vos, A., Rentergem, Y.V.: Reversible computing: from mathematical group theory to electronical circuit experiment. In: Proceedings of the Second Conference on Computing Frontiers, 2005, Ischia, Italy, May 4–6, 2005, pp. 35–44 (2005)

7. Draper, T.G., Kutin, S.A., Rains, E.M., Svore, K.M.: A logarithmic-depth quantum carry-lookahead adder. arXiv:quant-ph/0406142 (2008)
8. Green, A.S., Lumsdaine, P.L.F., Ross, N.J., Selinger, P., Valiron, B.: An introduction to quantum programming in quipper. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 110–124. Springer, Heidelberg (2013)
9. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: a scalable quantum programming language. In: Conference on Programming Language Design and Implementation, PLDI 2013, pp. 333–342. ACM (2013)
10. Jordan, S.P.: Strong equivalence of reversible circuits is coNP-complete. Quantum Information & Computation **14**(15–16), 1302–1307 (2014)
11. Kaarsgaard, R.: Towards a propositional logic for reversible logic circuits. In: de Haan, R. (ed.) Proceedings of the ESSLLI 2014 Student Session, pp. 33–41 (2014). http://www.kr.tuwien.ac.at/drm/dehaan/stus2014/proceedings.pdf
12. Miller, D.M., Maslov, D., Dueck, G.W.: A transformation based algorithm for reversible logic synthesis. In: Design Automation Conference, DAC, pp. 318–323 (2003)
13. Shende, V.V., Prasad, A.K., Markov, I.L., Hayes, J.P.: Synthesis of reversible logic circuits. IEEE Trans. on CAD of Integrated Circuits and Systems **22**(6), 710–722 (2003)
14. Soeken, M., Thomsen, M.K.: White dots *do* matter: rewriting reversible logic circuits. In: Dueck, G.W., Miller, D.M. (eds.) RC 2013. LNCS, vol. 7948, pp. 196–208. Springer, Heidelberg (2013)
15. Soeken, M., Thomsen, M.K., Dueck, G.W., Miller, D.M.: Self-inverse functions and palindromic circuits. arXiv 1502.05825 (2015)
16. Storme, L., De Vos, A., Jacobs, G.: Group theoretical aspects of reversible logic gates. J. UCS **5**(5), 307–321 (1999)
17. Takahashi, Y., Kunihiro, N.: A fast quantum circuit for addition with few qubits. Quantum Info. Comput. **8**(6), 636–649 (2008)
18. Thomsen, M.K.: Describing and optimising reversible logic using a functional language. In: Gill, A., Hage, J. (eds.) IFL 2011. LNCS, vol. 7257, pp. 148–163. Springer, Heidelberg (2012)
19. Thomsen, M.K., Axelsen, H.B.: Parallelization of reversible ripple-carry adders. Parallel Processing Letters **19**(1), 205–222 (2009)
20. Vedral, V., Barenco, A., Ekert, A.: Quantum networks for elementary arithmetic operations. Physical Review A **54**(1), 147–153 (1996)
21. Wille, R., Soeken, M., Miller, D.M., Drechsler, R.: Trading off circuit lines and gate costs in the synthesis of reversible logic. Integration, the VLSI Journal **47**(2), 284–294 (2014)
22. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In: Conference on Computing Frontiers, CF, pp. 43–54. ACM Press (2008)