A Library of Reversible Circuit Transformations (Work in Progress)

Christian Hutslar[1], Jacques Carette[2], and Amr Sabry[3]

[1] Indiana University chutslar@indiana.edu
[2] McMaster University carette@mcmaster.ca
[3] Indiana University sabry@indiana.edu

Abstract. Isomorphisms between finite types directly correspond to combinational, reversible, logical gates. Categorically they are morphisms in special classes of (bi-)monoidal categories. The coherence conditions for these categories determine sound and complete equivalences between isomorphisms. These equivalences were previously shown to correspond to a second-level of isomorphisms between the gate-modeling isomorphisms. In this work-in-progress report, we explore the use of that second level of isomorphisms to express semantic-preserving transformations and optimizations between reversible logical circuits. The transformations we explore are, by design, sound and complete therefore providing the basis for a complete library. Furthermore, we propose in future work, that attaching cost annotations to each level-2 transformation allows the development of strategies to transform circuits to optimal ones according to user-defined cost functions.

1 Introduction

Proving the equivalence of two combinational circuits is a common task. Most current approaches address it by proving the *extensional equivalence* of the circuits, i.e., by checking the equivalence of an exponential number of input-output pairs either directly or via an encoding to SAT [8, 6, 15]. We instead propose to apply our recent work [1] on type isomorphisms and their equivalences to this problem domain. Our approach produces equivalences of circuits using a sound and complete rewriting system with novel tradeoffs: the sizes of the proofs (i.e., rewriting sequences) are not necessarily proportional to the number of input-output pairs, thus directly exploiting any common structure between the circuits. Perhaps more interestingly, we propose future work in which the primitive rewriting steps can be annotated with cost parameters to rewrite circuits based on user-defined cost functions. We have currently implemented a naïve search strategy that can be used to prove some circuit equivalences and are experimenting with more advanced and more general strategies.

2 Π Family of Languages

Focusing on finite types, the building blocks of type theory are: the empty type (0), the unit type (1), the sum type (+), and the product (×) type. Before get-

$$
\begin{array}{rcl}
\mathit{id}{\leftrightarrow} : & \tau \leftrightarrow \tau & : \mathit{id}{\leftrightarrow} \\
\\
\mathit{unite}_{+}l : & 0 + \tau \leftrightarrow \tau & : \mathit{uniti}_{+}l \\
\mathit{swap}_{+} : & \tau_1 + \tau_2 \leftrightarrow \tau_2 + \tau_1 & : \mathit{swap}_{+} \\
\mathit{assocl}_{+} : & \tau_1 + (\tau_2 + \tau_3) \leftrightarrow (\tau_1 + \tau_2) + \tau_3 & : \mathit{assocr}_{+} \\
\\
\mathit{unite}_{*}l : & 1 * \tau \leftrightarrow \tau & : \mathit{uniti}_{*}l \\
\mathit{swap}_{*} : & \tau_1 * \tau_2 \leftrightarrow \tau_2 * \tau_1 & : \mathit{swap}_{*} \\
\mathit{assocl}_{*} : & \tau_1 * (\tau_2 * \tau_3) \leftrightarrow (\tau_1 * \tau_2) * \tau_3 & : \mathit{assocr}_{*} \\
\\
\mathit{absorbr} : & 0 * \tau \leftrightarrow 0 & : \mathit{factorzl} \\
\mathit{dist} : & (\tau_1 + \tau_2) * \tau_3 \leftrightarrow (\tau_1 * \tau_3) + (\tau_2 * \tau_3) & : \mathit{factor}
\end{array}
$$

$$
\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_2 \leftrightarrow \tau_3}{\vdash c_1 \odot c_2 : \tau_1 \leftrightarrow \tau_3}
$$

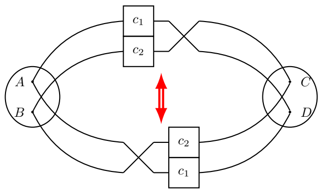
$$
\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \oplus c_2 : \tau_1 + \tau_3 \leftrightarrow \tau_2 + \tau_4}
\qquad
\frac{\vdash c_1 : \tau_1 \leftrightarrow \tau_2 \quad \vdash c_2 : \tau_3 \leftrightarrow \tau_4}{\vdash c_1 \otimes c_2 : \tau_1 * \tau_3 \leftrightarrow \tau_2 * \tau_4}
$$

Fig. 1. *Π*-terms and combinators.

ting into the formal theory, let's consider the intuition that types are modeled by (topological) spaces and that type equivalence is modeled by possible *deformations* of such spaces. Consider the space $(1+0)\times(1+1)$. This space is the product of two subspaces: the subspace $(1+0)$ which itself is the sum of the space 1 containing one point `tt` and the empty space 0 and the subspace $(1+1)$ which is the sum of two spaces each containing the one point `tt`. Any deformation of this space must at least preserve the number of points: we can neither create nor destroy points during any continuous deformation. Seeing that the number of points in our example space is 2, a reasonable hypothesis is that we can deform the space above to any other space with 2 points such as $1+1$ or $1+(1+0)$. What this really means is that we are treating the sum and product structure as malleable. Imagining a product structure as arranged in a grid, by stretching we can turn this structure to a sum structure arranged in a line, change the orientation of the grid by exchanging the axes, as well as other transformations that preserve the number of points using various "symmetries." These symmetries capture, in a stylized manner, a rich collection of space-time tradeoffs [12]. We formalize this intuition by saying that types form a *commutative semiring* (up to type isomorphism).

Another model of types is to view each type A as a collection of physical wires that can transmit $\|A\|$ distinct values where $\|A\|$ is the size of a type, computed as: $\|0\| = 0$; $\|1\| = 1$; $\|A+B\| = \|A\|+\|B\|$; and $\|A\times B\| = \|A\| * \|B\|$. Thus the type $\mathbb{B} = 1+1$ corresponds to a wire that can transmit two values, i.e., bits, and the type $\mathbb{B}\times\mathbb{B}\times\mathbb{B}$ corresponds to a collection of wires that can transmit three bits. From that perspective, a type isomorphism between types A and B (such that $\|A\| = \|B\| = n$) models a *reversible* combinational circuit that *permutes* the n different values. These type isomorphisms are collected in Fig. 1. We call the resulting language Π. It is known that these type isomorphisms are sound and complete for all permutations on finite types [4, 3] and hence that they are *complete* for expressing combinational circuits [5, 7, 13].

So far, the types encode conventional data structures, i.e, sets of values and structured trees of values and the isomorphisms act on such conventional data structures. Universal computation models however fundamentally rely on the fact that *programs are (or can be encoded as) data*, e.g., a Turing machine can be encoded as a string that another Turing machine (or even the same machine) can manipulate. In our setting, we ask whether the type isomorphisms in Fig. 1 can themselves be subject to (higher-level) reversible deformations? Before developing the theory, let's consider a small example consisting of two deformations between the types $A + B$ and $C + D$:



The top path is the Π program $(c_1 \oplus c_2) \odot \mathit{swap}_+$ which deforms the type A by c_1, deforms the type B by c_2, and deforms the resulting space by a twist that exchanges the two injections into the sum type. The bottom path performs the twist first and then deforms the type A by c_1 and the type B by c_2 as before. If one could imagine the paths as physical wires and the deformations c_1 and c_2 as arbitrary deformations on these wires then, holding the points A, B, C, and D fixed, it is possible to rotate the top part of the diagram to become identical to the bottom one. That rotation can be undone (reversed), which takes the bottom part of the diagram into the top part. In other words, there exists a deformation of the program $(c_1 \oplus c_2) \odot \mathit{swap}_+$ to the program $\mathit{swap}_+ \odot (c_2 \oplus c_1)$. We can also show that this means that, as permutations, $(c_1 \oplus c_2) \odot \mathit{swap}_+$ and $\mathit{swap}_+ \odot (c_2 \oplus c_1)$ are equal. This relation is non-trivial, as not all programs between the same types can be deformed into one another. The simplest example of inequivalent deformations are the two automorphisms of $1 + 1$, namely $id{\leftrightarrow}$ and swap_+.

Developing a collection of "templates" for such higher-level equivalences is reasonably straightforward (see for example the work of Miller et al [10]). To find a *complete* set of equivalences is much more difficult. The idea developed in our previous work [1] relies on "categorification" and a classical result by Laplaza [9]. We refer the interested reader to our previous work for the full set of *complete* "templates." A related idea [2] for a sound and complete set of equivalences on control-not gates is also developed in the context of the same monoidal categories as the ones we use suggesting that these categorical structures provide a common framework for reasoning about various classes of reversible circuits.

3 Reversible Circuit Equivalences

We illustrate our methodology using a small example. Consider a circuit that takes an input type consisting of three values $\widehat{a\ \widehat{b\ c}}$ and swaps the leftmost value with the rightmost value to produce $\widehat{c\ \widehat{b\ a}}$. We can implement two such circuits using our Agda library for Π:

```
swap-fl1 swap-fl2 : {a b c : U} → PLUS a (PLUS b c) ↔ PLUS c (PLUS b a)
swap-fl1 = assocl₊ ⊙ swap₊ ⊙ (id↔ ⊕ swap₊)

swap-fl2 = (id↔ ⊕ swap₊) ⊙
           assocl₊ ⊙
           (swap₊ ⊕ id↔) ⊙
           assocr₊ ⊙
           (id↔ ⊕ swap₊)
```

The first implementation rewrites the incoming values as follows:

$$\widehat{a\ \widehat{b\ c}} \to \widehat{\widehat{a\ b}\ c} \to \widehat{c\ \widehat{a\ b}} \to \widehat{c\ \widehat{b\ a}}\ .$$

The second implementation rewrites the incoming values as follows:

$$\widehat{a\ \widehat{b\ c}} \to \widehat{a\ \widehat{c\ b}} \to \widehat{\widehat{a\ c}\ b} \to \widehat{\widehat{c\ a}\ b} \to \widehat{c\ \widehat{a\ b}} \to \widehat{c\ \widehat{b\ a}}\ .$$

The two circuits are extensionally equal. Using the level-2 isomorphisms we can *explicitly* construct a sequence of rewriting steps that transforms the second circuit to the first. The proof can be read as follows: the first three lines "refocus" from a right-associated isomorphism onto the (left-associated) composition of the first 3 isomorphisms; then apply a complex rewrite on these (the "hexagon" coherence condition of symmetric braided monoidal categories); this exposes two inverse combinators next to each other — so we have to refocus on these to eliminate them; we finally re-associate to get the result.

```
swap-fl2⇔swap-fl1 : {a b c : U} → swap-fl2 {a} {b} {c} ⇔ swap-fl1
swap-fl2⇔swap-fl1 =
    ((id↔ ⊕ swap₊) ⊙ assocl₊ ⊙ (swap₊ ⊕ id↔) ⊙ assocr₊ ⊙ (id↔ ⊕ swap₊))
  ⇔⟨ id⇔ ⊡ assoc⊙l ⟩
    ((id↔ ⊕ swap₊) ⊙ (assocl₊ ⊙ (swap₊ ⊕ id↔)) ⊙ assocr₊ ⊙ (id↔ ⊕ swap₊))
  ⇔⟨ assoc⊙l ⟩
    (((id↔ ⊕ swap₊) ⊙ assocl₊ ⊙ (swap₊ ⊕ id↔)) ⊙ assocr₊ ⊙ (id↔ ⊕ swap₊))
  ⇔⟨ assoc⊙l ⊡ id⇔ ⟩
    ((((id↔ ⊕ swap₊) ⊙ assocl₊) ⊙ (swap₊ ⊕ id↔)) ⊙ assocr₊ ⊙ (id↔ ⊕ swap₊))
  ⇔⟨ hexagonl⊕r ⊡ id⇔ ⟩
    (((assocl₊ ⊙ swap₊) ⊙ assocl₊) ⊙ assocr₊ ⊙ (id↔ ⊕ swap₊))
  ⇔⟨ assoc⊙r ⟩
    ((assocl₊ ⊙ swap₊) ⊙ assocl₊ ⊙ assocr₊ ⊙ (id↔ ⊕ swap₊))
  ⇔⟨ id⇔ ⊡ assoc⊙l ⟩
    ((assocl₊ ⊙ swap₊) ⊙ (assocl₊ ⊙ assocr₊) ⊙ (id↔ ⊕ swap₊))
```

```
⇔⟨ id⇔ ⊡ (linv⊙l ⊡ id⇔) ⟩
  ((assocl₊ ⊙ swap₊) ⊙ id↔ ⊙ (id↔ ⊕ swap₊))
⇔⟨ id⇔ ⊡ idl⊙l ⟩
  ((assocl₊ ⊙ swap₊) ⊙ (id↔ ⊕ swap₊))
⇔⟨ assoc⊙r ⟩
  ((assocl₊ ⊙ swap₊ ⊙ (id↔ ⊕ swap₊)) □)
```

4 Circuit Examples and Cost Semantics

We can now apply our approach to more realistic examples from the literature. For example, consider the following figure from a paper by Shende et al [11]:

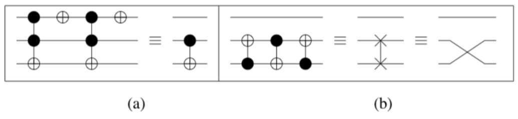


Figure 3: Reversible circuit equivalences: (a) $T^3_{1,2}\cdot N^1\cdot T^3_{1,2}\cdot N^1 = C^3_2$, (b) $C^2_3\cdot C^3_2\cdot C^2_3 = S^{2,3}$; subscripts identify "control bits" while superscripts identify bits whose values actually change.

The figure shows two equivalences (a) and (b) of circuits. We can express such circuits as (level-1) type isomorphisms *and* express the equivalences between them as (level-2) isomorphisms between type isomorphisms. The full code for the circuits in part (a) and their equivalence takes 2-3 pages of Agda code and is not included for lack of space. Even for such small circuits, the proofs are by no means "obvious", and require a good deal of experience to develop. The main difficulty is that it is often necessary to make the circuits larger in intermediate steps in order to expose some structure that can later be simplified. To aid in the development of larger proofs, we have implemented a simple search procedure that produces a list of candidate level-2 isomorphisms to apply at each step. We conjecture that techniques quite familiar from AI search could help prune the search space and produce "good" candidates for completing proofs.

We are additionally experimenting with user annotations that can guide the search. Each level-2 combinator can be annotated with various "cost" annotations indicating whether it reduces the number of gates, reduces the number of choice points, or other cost functions. Then one can ask for a proof that takes no more than a certain number of steps or a proof that does not create more than a certain number of additional wires etc. We illustrate these ideas by defining a simple cost function and using it to annotate level-2 combinators.

We define the *length* $L(c)$ of a composite circuit c as follows: the length of a sequential composition of circuits is the sum of the lengths of the subcircuits $L(f \odot g) = L(f) + L(g)$; and the length of choice or parallel composition is the maximum of either branch $L(f \oplus g) = L(f \otimes g) = \max(L(f), L(g))$. For primitive gates, the length needs to be postulated to reflect the "length" of the computation involved in applying that primitive. As examples, consider the following two level-2 combinators:

```
linv⊙l' : {t₁ t₂ : U} {c : t₁ ↔ t₂} → (c ⊙ ! c) ⇔ id↔
idl⊙l'  : {t₁ t₂ : U} {c : t₁ ↔ t₂} → (id↔ ⊙ c) ⇔ c
```

Assuming that $id \leftrightarrow$ takes a unit length of computation, the first can be annotated with $L(c) * 2 \Leftrightarrow 1$ and the second with $L(c) + 1 \Leftrightarrow L(c)$ indicating that the first combinator reduces the length of the circuit from twice the length of c to 1 and the second combinator reduces the length of the circuit by 1. Such annotations can then be used to constrain or guide the search for transformations between circuits.

5 Conclusion and Future Work

We propose a purely algebraic perspective for reasoning about the equivalence of reversible circuits. Our approach is founded on deep ideas from category theory and includes a *complete* set of rewrite rules, i.e., if two circuits are equivalent then there exists a sequence of rewriting steps using the level-2 isomorphisms from one circuit to the other. Searching for one such sequence is difficult but can benefit from the well-developed AI search technology and from possible user annotations to constrain and guide the search. It is possible to define several "canonical" representations of circuits, e.g., sequences of transpositions, and have a deterministic algorithm for reducing circuits to these canonical representations. These approaches are however typically computationally expensive [14] and might not produce an effective procedure for deciding circuit equivalence using further heuristics.

Acknowledgments. We thank the anonymous reviewers for their valuable comments and Kyle Carter for insights on the cost semantics.

References

1. Jacques Carette and Amr Sabry. Computing with Semirings and Weak Rig Groupoids. In *ESOP*, volume 9632 of *Lecture Notes in Computer Science*, pages 123–148. Springer, 2016.
2. J. Robin B. Cockett, Cole Comfort, and Priyaa Srinivasan. The category CNOT. In Bob Coecke and Aleks Kissinger, editors, *Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017.*, volume 266 of *EPTCS*, pages 258–293, 2017.
3. M. P. Fiore, R. Di Cosmo, and V. Balat. Remarks on isomorphisms in typed calculi with empty and sum types. *Annals of Pure and Applied Logic*, 141(1-2):35–50, 2006.
4. Marcelo Fiore. Isomorphisms of generic recursive polynomial types. In *POPL*, pages 77–88. ACM, 2004.
5. E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, 1982.
6. Eugene Goldberg and Yakov Novikov. How good can a resolution based SAT-solver be? In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 37–52, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

7. Roshan P. James and Amr Sabry. Information effects. In *POPL*, pages 73–84. ACM, 2012.
8. A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai. Robust boolean reasoning for equivalence checking and functional property verification. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 21(12):1377–1394, November 2006.
9. Miguel L. Laplaza. Coherence for distributivity. In G.M. Kelly, M. Laplaza, G. Lewis, and Saunders Mac Lane, editors, *Coherence in Categories*, volume 281 of *Lecture Notes in Mathematics*, pages 29–65. Springer Verlag, Berlin, 1972.
10. D. Michael Miller, Dmitri Maslov, and Gerhard W. Dueck. A transformation based algorithm for reversible logic synthesis. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 318–323, New York, NY, USA, 2003. ACM.
11. V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes. Synthesis of reversible logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6):710–722, June 2003.
12. Zachary Sparks and Amr Sabry. Superstructural reversible logic. In *3rd International Workshop on Linearity*, 2014.
13. Tommaso Toffoli. Reversible computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 632–644. Springer-Verlag, 1980.
14. Siyao Xu. *Reversible Logic Synthesis with Minimal Usage of Ancilla Bits.* PhD thesis, MIT, 2015.
15. Shigeru Yamashita and Igor L. Markov. Fast equivalence-checking for quantum circuits. In *Proceedings of the 2010 IEEE/ACM International Symposium on Nanoscale Architectures*, NANOARCH '10, pages 23–28, Piscataway, NJ, USA, 2010. IEEE Press.