Symmetries in Reversible Programming

From Symmetric Rig Groupoids to Reversible Programming Languages

VIKRAMAN CHOUDHURY, Indiana University, USA and University of Cambridge, UK
JACEK KARWOWSKI, University of Warsaw, Poland
AMR SABRY, Indiana University, USA

The Π family of reversible programming languages for boolean circuits is presented as a syntax of combinators witnessing type isomorphisms of algebraic data types. In this paper, we give a denotational semantics for this language, using weak groupoids à la Homotopy Type Theory, and show how to derive an equational theory for it, presented by 2-combinators witnessing equivalences of type isomorphisms.

We establish a correspondence between the syntactic groupoid of the language and a formally presented univalent subuniverse of finite types. The correspondence relates 1-combinators to 1-paths, and 2-combinators to 2-paths in the universe, which is shown to be sound and complete for both levels, forming an equivalence of groupoids. We use this to establish a Curry-Howard-Lambek correspondence between Reversible Logic, Reversible Programming Languages, and Symmetric Rig Groupoids, by showing that the syntax of Π is presented by the free symmetric rig groupoid, given by finite sets and bijections.

Using the formalisation of our results, we perform normalisation-by-evaluation, verification and synthesis of reversible logic gates, motivated by examples from quantum computing. We also show how to reason about and transfer theorems between different representations of reversible circuits.

CCS Concepts: • **Theory of computation** → **Type theory**; **Categorical semantics**; **Denotational semantics**; • **Mathematics of computing** → **Combinatorics**; • **Software and its engineering** → **Formal methods**; • **Computing methodologies** → **Symbolic and algebraic manipulation**.

Additional Key Words and Phrases: reversible computing, reversible programming languages, homotopy type theory, univalent foundations, groupoids, groups, rewriting, permutations, type isomorphisms

ACM Reference Format:
Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. 2022. Symmetries in Reversible Programming: From Symmetric Rig Groupoids to Reversible Programming Languages. *Proc. ACM Program. Lang.* 6, POPL, Article 6 (January 2022), 32 pages. https://doi.org/10.1145/3498667

1 INTRODUCTION

Consider two programs that implement the *same* permutation using different sequences of type isomorphisms:

$$A + (B + C) \leftrightarrow (A + B) + C \leftrightarrow C + (A + B) \leftrightarrow C + (B + A)$$
$$A + (B + C) \leftrightarrow A + (C + B) \leftrightarrow (A + C) + B \leftrightarrow (C + A) + B \leftrightarrow C + (A + B) \leftrightarrow C + (B + A)$$

Authors' addresses: Vikraman Choudhury, Department of Computer Science, Indiana University, Bloomington, 47408, USA, vikraman@indiana.edu, Department of Computer Science and Technology and University of Cambridge, Cambridge, CB3 0FD, UK, vc378@cam.ac.uk; Jacek Karwowski, Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warsaw, 00-927, Poland, jk359713@students.mimuw.edu.pl; Amr Sabry, Department of Computer Science, Indiana University, Bloomington, 47408, USA, sabry@indiana.edu.

2475-1421/2022/1-ART6
https://doi.org/10.1145/3498667

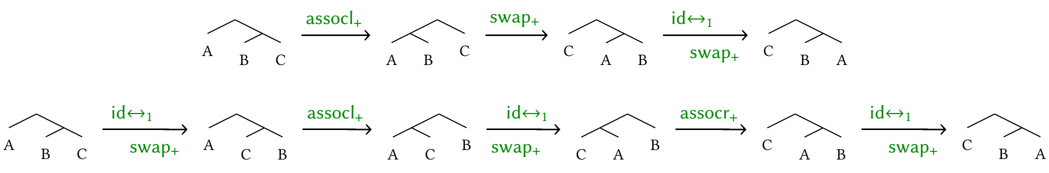


Fig. 1. Type isomorphisms as tree transformations

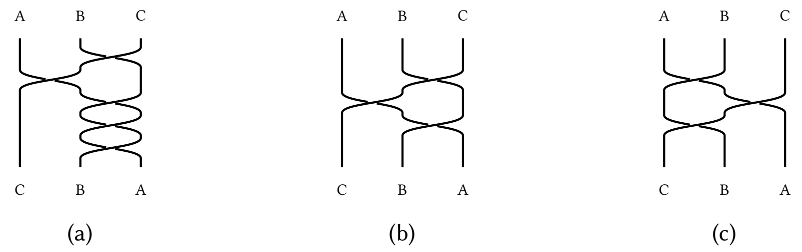


Fig. 2. Type isomorphisms as braiding diagrams

These permutations are written as sequences of primitive operations: associativity, symmetry, and composition. Can we find necessary and sufficient equations to identify *all* such equivalent sequences of type isomorphisms?

Such sequences of type isomorphisms are pervasive in reversible boolean circuits, which are at the core of quantum computing. Typically, they might be formalised as permutations $\{0,1\}^k \to \{0,1\}^k$ on bit strings of some length k [Aaronson et al. 2017]. But from the perspective of programming languages, permutations and reversible circuits can be more conveniently expressed as isomorphisms over algebraic data types, as proposed in the Π family of reversible languages [James and Sabry 2012, 2014]. The syntax of Π is inspired by the sound and complete axiomatisation of type isomorphisms of lambda calculi [Fiore et al. 2002], with respect to the bicartesian structure, that is, product and sum types. Π programs correspond to type isomorphisms of finite types, and in this paper, we make the observation that the syntax of Π is a presentation of the free symmetric rig groupoid (on zero generators).

It is folklore that the groupoid of finite sets and bijections is the free symmetric rig groupoid on zero generators [Baez and Dolan 2001; Kelly 1974; Laplaza 1972]. Our main result formally establishes this correspondence by giving an equational theory for Π that includes exactly all the necessary equations to decide equivalence of Π programs. As conjectured by Carette and Sabry [2016], these equations correspond to the *coherence conditions of symmetric rig groupoids*, answering the conjecture in the positive.

Equivalence by Example. Without going into details of proofs, let us try to answer the question for the original pair of examples.

The first step is to realise that the use of associativity is uninteresting, and the only steps with interesting computational content are the swaps. The swaps can be either *small* or *big* – they can happen between leaf nodes, or between bigger subtrees, respectively, as shown in Figure 1.

First, we normalise the types to a list-like representation $A + (B + (C + \mathbb{0}))$, where $\mathbb{0}$ is the empty type. In this representation, each type is identified with its index, A has index 0, B has index 1, and C has index 2. Then, we compile each primitive isomorphism to a list of adjacent transpositions, which are composed by appending the lists. A small swap is trivially implemented as a single adjacent transposition. To compile a big swap into adjacent transpositions, we traverse the subtrees in-order and recursively swap elements from the left by transposing them across the ones in the middle – this generates a large number of adjacent transpositions. For the two programs, we get the following two lists: $[1, 0, 1, 1, 1]$ (Figure 2a) and $[1, 0, 1]$ (Figure 2b), where the number k encodes a transposition of elements at indices k and $k + 1$.

Swapping is symmetric, that is, swapping two elements and immediately swapping them back produces the same permutation. The first list contains such redundant operations, so to eliminate these, we will normalise the lists, by setting up reduction relations and an appropriate rewriting system. This system normalises both lists to $[0, 1, 0]$ (Figure 2c), which is lexicographically the smallest list that corresponds to this permutation. Since both programs have the same normal form, they're equivalent!

The crux of designing an equational theory for Π relies on the choice of relations we use to design this rewriting system. First, there should be enough equations to compile programs to their normalised forms, that is, sequences of adjacent transpositions, and second, there should be enough equations to reduce them to a normal form, that is, the reduced list of transpositions. We will show that these equations correspond to the coherence conditions for symmetric rig groupoids.

From the normalised list $[0, 1, 0]$, we can recover a permutation on a list of 3 elements $[a, b, c]$ as follows. We think of it as insertion-sorting the elements of the list. Starting from the list $[a, b, c]$, we first insert b at the right place – by applying transposition 0, we get the list $[b, a, c]$. Then, element c is inserted in the right place by applying transpositions 1 and 0 – as a result, we get the desired permutation $[c, b, a]$. Notice that we could specify a more compact way of describing this process, since the key information was only how many shifts we needed to apply to an element. We could describe this procedure using a code $(0, 1, 2)$, which says how many inversions to apply to elements a, b, and c, respectively.

Using this algorithm, we can turn a type isomorphism into a bijection of finite sets, relating the operational semantics of Π as permutations of finite sets of bits, to our denotational semantics. This allows us to establish full abstraction and adequacy.

Outline and Contributions. Our main result is a proof of the soundness and completeness of Π with respect to its semantics in the weak symmetric rig groupoid of finite sets, bijections, and homotopies, formalised in HoTT/UF. We state our result as a Curry-Howard-Lambek correspondence for Reversible Logic [Sparks and Sabry 2014], Reversible Programming Languages [James and Sabry 2012], and Symmetric Rig Groupoids [Carette and Sabry 2016; Kelly 1974; Laplaza 1972], using which we can build a toolbox of technical devices for reasoning about reversible circuits.

- We start in Section 2 by presenting a few reversible circuits in the popular IBM Qiskit framework [Aleksandrowicz et al. 2019] to serve as running examples throughout the paper.
- In Section 3, we introduce the two-level language Π [Carette and Sabry 2016] and illustrate how to write reversible circuits and their equivalences using 1-combinators and 2-combinators respectively. We give a semantic account of the language by translating each level-1 program to a bijection between finite sets, and verifying that programs identified by level-2 constructs denote the same bijection.
- Section 4 describes the construction of $\mathcal{U}_{\mathsf{Fin}}$, the groupoid of finite sets and bijections, in Homotopy Type Theory and Univalent Foundations (HoTT/UF). We define and characterise the notion of a univalent subuniverse, and construct $\mathcal{U}_{\mathsf{Fin}}$ as a univalent subuniverse which classifies all finite types. We establish that paths in $\mathcal{U}_{\mathsf{Fin}}$ are families of loops on finite sets of specified cardinality, given by $\mathsf{Aut}(\mathsf{Fin}_n)$, which produces the group of permutations on Fin_n.

- In Section 5, we proceed to give a presentation of this group, as the symmetric group $\mathsf{S_n}$ with generators and relations, and solve its word problem. In particular, we present $\mathsf{S_n}$ as a Coxeter group, build a rewriting system based on Coxeter relations, and prove confluence and termination. Using our rewriting system, we establish that normal forms for words in $\mathsf{S_n}$ are equivalent to Lehmer codes [Lehmer 1960], which are a convenient and compact representation of permutations. Finally, we show that there is an equivalence between Lehmer codes and permutations $\mathsf{Aut}(\mathsf{Fin_n})$ given by the Lehmer encode-decode algorithm.
- In Section 6, we show how to interpret the language Π into the groupoid $\mathcal{U}_{\mathsf{Fin}}$, in stages. First, we define a subset Π^+ of the language which includes the additive monoidal structure, and show how to translate Π programs to Π^+ programs. Then, we further define a normalised form for this language called $\Pi^\wedge$, which has normalised 1-combinators and 2-combinators corresponding to adjacent transpositions. We show that Π^+ can be translated to $\Pi^\wedge$ and back. Then, we show how to interpret this language $\Pi^\wedge$ into $\mathcal{U}_{\mathsf{Fin}}$ – the 1-combinators are translated into permutations via words in $\mathsf{S_n}$, and 2-combinators are interpreted as 2-paths in $\mathcal{U}_{\mathsf{Fin}}$. We further show how to quote back a permutation in $\mathcal{U}_{\mathsf{Fin}}$ into a 1-combinator using the normal forms for words in $\mathsf{S_n}$. The main result of this section is a symmetric monoidal equivalence between the syntactic groupoids of Π^+ and $\Pi^\wedge$, with $\mathcal{U}_{\mathsf{Fin}}$. Finally, we also establish full abstraction and adequacy of this model with respect to the operational semantics.
- In Section 7, we show applications of our results to reversible circuits, using our formalisation. Our results are stated using HoTT/UF [Univalent Foundations Program 2013], and formalised using the HoTT-Agda library (around 7,500 lines of code). Using the formalisation, we are able to extract procedures for: (1) the synthesis of a reversible circuit from a permutation on a finite set, (2) the verification that a reversible circuit realises a given permutation on finite sets, (3) a normalisation-by-evaluation (NbE) procedure that reduces reversible circuits to canonical normal forms, (4) a sound and complete calculus for reasoning about reversible circuits and their equivalences, and (5) the transfer of theorems about permutations and reversible circuits from one representation to the other.

The proofs of some of our lemmas and propositions and theorems, as well as additional material, are given in the supplementary appendices in the extended version [Choudhury, Karwowski, and Sabry 2021b], and we refer to them in the text. Our accompanying Agda code [Choudhury, Karwowski, and Sabry 2021a] contains the formalisation of the full syntax, and most of our proofs.

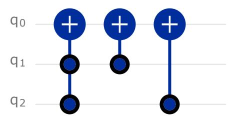
2 REVERSIBLE CIRCUITS IN QISKIT

Classical reversible boolean circuits are at the core of most quantum algorithms and hence are supported by popular platforms for quantum computing such as IBM Qiskit [Aleksandrowicz et al. 2019]. Specifically, the Qiskit framework provides the following universal set of gates for reversible computing: not (boolean negation, called `x`), cnot (conditional negation of the second input if the first is true; called `cx`), and toffoli (conditional negation of the third input if both the first two inputs are true; called `ccx`) gates. Additionally, Qiskit allows implicit re-shuffling of bits by allowing each operation to specify the indices of its input bits.

For concreteness, we demonstrate two different circuits that implement the following reversible function specification $\mathsf{reversibleOr}(h, b_1, b_2) = (h \veebar (b_1 \vee b_2),\ b_1,\ b_2)$ where $\vee$ is disjunction and $\veebar$ is exclusive-disjunction. The circuits are presented in both the textual interface `qasm` and the graphical interface, in Figure 3.

There is a wealth of manual and algorithmic approaches for producing circuits such as these two [Maslov 2003; Shende et al. 2003]. The circuit on the left was manually produced using a standard synthesis algorithm for reversible circuits [Miller et al. 2003]. The circuit on the right

```
ccx q[1], q[2], q[0];
cx  q[1], q[0];
cx  q[2], q[0];
```



```
cx  q[1], q[0];
x   q[1];
ccx q[1], q[2], q[0];
x   q[1];
```

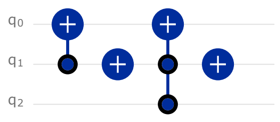


Fig. 3. reversibleOr in Qiskit

was produced using an approach that analyzes the recursive structure of reversibleOr (and would generalise to computing the disjunction of more than two inputs).

From the specification of the circuit, we expect input `011` to be mapped to `111`. To gain some intuition, we trace the evaluation of each circuit for input `011`. In this context, the most significant bit is at index 0. In the first circuit, the `ccx` gate negates `q[0]` since both `q[1]` and `q[2]` are true producing `111`; the following `cx` gate produces `011`; finally the last `cx` produces the result `111`. For the second circuit, the trace of the execution on the same input value `011` goes through the stages `111`, `101`, `101`, and finally `111`.

Although rather trivial, the examples above illustrate the general idea. A more interesting example would be the classical core of Shor's algorithm which requires a circuit implementing modular exponentiation $f(r) = a^r \bmod N$ for fixed a and N. Using Toffoli's construction [1980], a reversible specification of this function is $f^R(r, h) = (r, f(r) \veebar h)$ where the exclusive disjunction $\veebar$ is applied bitwise. When $h = 0$ the second component of the result of f^R is the result of the original modular exponentiation f. However, as explained in standard accounts of the algorithm (e.g., the Qiskit implementation), producing an efficient modular exponentiation circuit from this specification is not straightforward and is actually the bottleneck in Shor's algorithm. Typical derivations of the circuit start from elementary gates, build a circuit for reversible disjunction (like the two circuits above), reversible conjunction, a circuit for a half-adder, a circuit for computing the carry, progressing to a circuit for modular addition, which is used to build a circuit for modular multiplication, and then finally a circuit for modular exponentiation taking care at each step to avoid the exponential blowup (e.g., by implementing exponentiation by squaring instead of repeated multiplication) [Beauregard 2003].

3 A REVERSIBLE PROGRAMMING LANGUAGE

The circuit model of reversible computation discussed in the previous section is a useful abstraction close to the hardware platform. However, since its main data abstraction is a *sequence of wires*, it only provides an "assembly-level" programming abstraction (e.g., `qasm`). As motivated by Lafont [2003], a mathematical model based on permutations of finite sets provides a richer algebraic structure, using which we describe a reversible programming language in this section.

3.1 The Π Family of Languages

In reversible boolean circuits, the number of input bits matches the number of output bits. Thus, a key insight for a programming language of reversible circuits is to ensure that each primitive operation preserves the number of bits, which is just a natural number. The algebraic structure of natural numbers as the free commutative semiring (or, commutative rig), with $(0, +)$ for addition, and $(1, \times)$ for multiplication then provides sequential, vertical, and horizontal circuit composition.

$$
\begin{array}{rrcll}
\mathsf{id}{\leftrightarrow_1} : & A & \leftrightarrow_1 & A & : \mathsf{id}{\leftrightarrow_1} \\[4pt]
\mathsf{unite_+l} : & \mathbb{0} + A & \leftrightarrow_1 & A & : \mathsf{uniti_+l} \\
\mathsf{swap_+} : & A + B & \leftrightarrow_1 & B + A & : \mathsf{swap_+} \\
\mathsf{assocl_+} : & A + (B + C) & \leftrightarrow_1 & (A + B) + C & : \mathsf{assocr_+} \\[4pt]
\mathsf{unite{\star}l} : & \mathbb{1} \times A & \leftrightarrow_1 & A & : \mathsf{uniti{\star}l} \\
\mathsf{swap{\star}} : & A \times B & \leftrightarrow_1 & B \times A & : \mathsf{swap{\star}} \\
\mathsf{assocl{\star}} : & A \times (B \times C) & \leftrightarrow_1 & (A \times B) \times C & : \mathsf{assocr{\star}} \\[4pt]
\mathsf{absorbr} : & \mathbb{0} \times A & \leftrightarrow_1 & \mathbb{0} & : \mathsf{factorzl} \\
\mathsf{dist} : & (A + B) \times C & \leftrightarrow_1 & (A \times C) + (B \times C) & : \mathsf{factor}
\end{array}
$$

$$
\frac{\vdash c_1 : A \leftrightarrow_1 B \quad \vdash c_2 : B \leftrightarrow_1 C}{\vdash c_1 \circledcirc c_2 : A \leftrightarrow_1 C}
\qquad
\frac{\vdash c_1 : A \leftrightarrow_1 B \quad \vdash c_2 : C \leftrightarrow_1 D}{\vdash c_1 \oplus c_2 : A + C \leftrightarrow_1 B + D}
\qquad
\frac{\vdash c_1 : A \leftrightarrow_1 B \quad \vdash c_2 : C \leftrightarrow_1 D}{\vdash c_1 \otimes c_2 : A \times C \leftrightarrow_1 B \times D}
$$

Fig. 4. Π-terms, combinators, and their types.

These commutative rig identities can be used to design a logic for reversible programming [James and Sabry 2012; Sparks and Sabry 2014]. To interpret natural number identities as reversible programs, the logic needs to be equipped with values and types, and a notion of operational semantics and contextual equivalence, giving a computational interpretation of the commutative rig structure [James and Sabry 2012]. On the semantic side, the natural space to consider is the groupoidification of a commutative rig, that is, a symmetric rig groupoid.

Putting these ideas together, the programming language *Π*, whose syntax is given below, embodies the computational content of isomorphisms of finite types, or permutations.

$$
\begin{array}{llcl}
\textit{Value types} & A, B, C, D & ::= & \mathbb{0} \mid \mathbb{1} \mid A + B \mid A \times B \\
\textit{Values} & v, w, x, y & ::= & \mathsf{tt} \mid \mathsf{inj_1}\ v \mid \mathsf{inj_2}\ v \mid (v, w) \\
\textit{Program types} & & & A \leftrightarrow_1 B \\
\textit{Programs} & c & ::= & \text{(See Figure 4)}
\end{array}
$$

The language of types is built from the empty type ($\mathbb{0}$), the unit type ($\mathbb{1}$) containing just one value tt, the sum type (+) containing values of the form $\mathsf{inj_1}\ v$ and $\mathsf{inj_2}\ v$, and the product type (×) containing pairs of values (v_1, v_2).

To see how this language expresses reversible circuits, we present a few examples using the Agda embedding of the language [Choudhury et al. 2021a; Sabry et al. 2021]. First it is possible to directly mimic the qasm-perspective by defining types that describe sequences of booleans ($\mathbb{2}^n$). We use the type $\mathbb{2} = \mathbb{1} + \mathbb{1}$ to represent booleans with $\mathsf{inj_1}\ \mathsf{tt}$ representing true, and $\mathsf{inj_2}\ \mathsf{tt}$ representing false. Boolean negation (the x-gate) is straightforward to define using the primitive combinator $\mathsf{swap_+}$. We can represent *n*-bit words using an n-ary product of boolean values, thus the type $\mathbb{2} \times (\mathbb{2} \times \mathbb{2})$ (abbreviated $\mathbb{B}\ 3$) corresponds to a collection of wires that can transmit three bits. To express the cx and ccx gates, we need to encode a notion of conditional expression. Such conditionals turn out to be expressible using the distributivity and factoring identities of rigs as shown below:

```
cif : (c₁ c₂ : A ↔₁ A) → (𝟚 × A ↔₁ 𝟚 × A)
cif c₁ c₂ = dist ⊚ ((id↔₁ ⊗ c₁) ⊕ (id↔₁ ⊗ c₂)) ⊚ factor
```

The input value of type $\mathbb{2} \times A$ is processed by the distribute operator dist, which converts it into a value of type $(\mathbb{1} \times A) + (\mathbb{1} \times A)$. In the left branch, which corresponds to the case when the boolean is true, the combinator c_1 is applied to the value of type A. The right branch, which corresponds to the boolean being false, passes the value of type A through the combinator c_2. The inverse of dist, namely factor is applied to get the final result. Using this conditional operator, cx is defined as cif x id↔₁ and ccx is defined as cif cx id↔₁. With these conventions, the first circuit in the previous section is transcribed as follows:

```
C[BA]-[CA]B : C × (B × A) ↔₁ (C × A) × B
C[BA]-[CA]B = (id↔₁ ⊗ swap⋆) ⊚ assocl⋆

reversibleOr1 : 𝔹 3 ↔₁ 𝔹 3
reversibleOr1 = A[BC]-C[BA] ⊚ ccx ⊚ (id↔₁ ⊗ cx) ⊚ C[BA]-[CA]B ⊚ (cx ⊗ id↔₁) ⊚ [CA]B-A[BC]
```

where we clearly see the sequences of the three operations ccx, cx, and cx but, instead of using the indices in the sequence of wires to identify the relevant parameters, here we use structural isomorphisms to re-shuffle the types. We only show one of these re-shuffling isomorphisms and elide the others. For the second circuit, instead of transcribing it directly, we express it using cif.

```
reversibleOr2 : 𝔹 3 ↔₁ 𝔹 3
reversibleOr2 = A[BC]-B[CA] ⊚ cif (x ⊗ id↔₁) cx ⊚ B[CA]-A[BC]
```

Like the original circuit, we examine the bit at index 1 (corresponding to the component B in a tuple $(A, (B, C))$): if the bit is true, we perform an x operation on component A, and otherwise we perform a cx operation on (C, A). The two uses of x gates in the circuit are now unnecessary as they were only needed to encode a two-way conditional expression using a sequence of one-way conditional expressions (the only ones available in the linear circuit model).

All of this is only half the story, however. A sound semantics for Π using weak rig groupoids was established by Carette and Sabry [2016], and conjectured to be complete. For this semantics, *coherence conditions* for symmetric rig groupoids that identify different syntactic representations of the same permutation [Laplaza 1972], were collected in a second level of Π syntax as level-2 combinators. Each level-2 combinator is of the form $c_1 \leftrightarrow_2 c_2$ for appropriate c_1 and c_2 of the same level-1 type $A \leftrightarrow_1 B$ and asserts that c_1 and c_2 denote the same bijection. For example, we have the following level-2 combinators dealing with associativity:

```
assoc⊚l  : {c₁ : A ↔₁ B} {c₂ : B ↔₁ C} {c₃ : C ↔₁ D} → (c₁ ⊚ (c₂ ⊚ c₃)) ↔₂ ((c₁ ⊚ c₂) ⊚ c₃)
assoc⊚r  : {c₁ : A ↔₁ B} {c₂ : B ↔₁ C} {c₃ : C ↔₁ D} → ((c₁ ⊚ c₂) ⊚ c₃) ↔₂ (c₁ ⊚ (c₂ ⊚ c₃))
assocl₊l : {c₁ : A ↔₁ B} {c₂ : C ↔₁ D} {c₃ : E ↔₁ F} → ((c₁ ⊕ (c₂ ⊕ c₃)) ⊚ assocl₊) ↔₂ (assocl₊ ⊚ ((c₁ ⊕ c₂) ⊕ c₃))
assocl₊r : {c₁ : A ↔₁ B} {c₂ : C ↔₁ D} {c₃ : E ↔₁ F} → (assocl₊ ⊚ ((c₁ ⊕ c₂) ⊕ c₃)) ↔₂ ((c₁ ⊕ (c₂ ⊕ c₃)) ⊚ assocl₊)
assocr₊r : {c₁ : A ↔₁ B} {c₂ : C ↔₁ D} {c₃ : E ↔₁ F} → (((c₁ ⊕ c₂) ⊕ c₃) ⊚ assocr₊) ↔₂ (assocr₊ ⊚ (c₁ ⊕ (c₂ ⊕ c₃)))
assocr₊l : {c₁ : A ↔₁ B} {c₂ : C ↔₁ D} {c₃ : E ↔₁ F} → (assocr₊ ⊚ (c₁ ⊕ (c₂ ⊕ c₃))) ↔₂ (((c₁ ⊕ c₂) ⊕ c₃) ⊚ assocr₊)
```

The full set of level-2 combinators is large; the remaining combinators are listed in Appendix A.1 in the extended version [Choudhury et al. 2021b], and also in the accompanying Agda formalisation [Choudhury et al. 2021a].

3.2 Semantics

Below we present a simple denotational semantics for our language, using finite types and type isomorphisms. Each Π type A is mapped to a (finite) set $[\![A]\!]$ and each combinator $c : A \leftrightarrow_1 B$ is mapped to a (bijective) function $[\![A]\!] \to [\![B]\!]$. We describe this semantics by writing an interpreter in Agda. First, we state the semantics for types, where $\bot$ is the empty set, $\top$ is the singleton set, $\sqcup$ is the disjoint union of sets, and $\times$ is the cartesian product of sets.

$$\begin{array}{rcl} [\![\mathbb{0}]\!] & = & \bot \\ [\![\mathbb{1}]\!] & = & \top \\ [\![A + B]\!] & = & [\![A]\!] \sqcup [\![B]\!] \\ [\![A \times B]\!] & = & [\![A]\!] \times [\![B]\!] \end{array}$$

For combinators, we show explicitly how values reduce along each combinator, similar to a big-step operational semantics [Chen and Sabry 2021; James and Sabry 2014].

$$
\begin{array}{lll}
[\![\, \mathsf{unite_{+}l} \,]\!]\ (\mathsf{inl}\, v) & = & v \\
[\![\, \mathsf{uniti_{+}l} \,]\!]\ v & = & \mathsf{inl}\, v \\
[\![\, \mathsf{swap_{+}} \,]\!]\ (\mathsf{inl}\, v) & = & \mathsf{inr}\, v \\
[\![\, \mathsf{swap_{+}} \,]\!]\ (\mathsf{inr}\, v) & = & \mathsf{inl}\, v \\
[\![\, \mathsf{assocl_{+}} \,]\!]\ (\mathsf{inl}\, v) & = & \mathsf{inl}\,(\mathsf{inl}\, v) \\
[\![\, \mathsf{assocl_{+}} \,]\!]\ (\mathsf{inr}\,(\mathsf{inl}\, v)) & = & \mathsf{inl}\,(\mathsf{inr}\, v) \\
[\![\, \mathsf{assocl_{+}} \,]\!]\ (\mathsf{inr}\,(\mathsf{inr}\, v)) & = & \mathsf{inr}\, v \\
[\![\, \mathsf{assocr_{+}} \,]\!]\ (\mathsf{inl}\,(\mathsf{inl}\, v)) & = & \mathsf{inl}\, v \\
[\![\, \mathsf{assocr_{+}} \,]\!]\ (\mathsf{inl}\,(\mathsf{inr}\, v)) & = & \mathsf{inr}\,(\mathsf{inl}\, v) \\
[\![\, \mathsf{assocr_{+}} \,]\!]\ (\mathsf{inr}\, v) & = & \mathsf{inr}\,(\mathsf{inr}\, v)
\end{array}
\qquad
\begin{array}{lll}
[\![\, \mathsf{unite\star l} \,]\!]\ (\star, v) & = & v \\
[\![\, \mathsf{uniti\star l} \,]\!]\ v & = & (\star, v) \\
[\![\, \mathsf{swap\star} \,]\!]\ (v_1, v_2) & = & (v_2, v_1) \\
[\![\, \mathsf{assocl\star} \,]\!]\ (v_1, (v_2, v_3)) & = & ((v_1, v_2), v_3) \\
[\![\, \mathsf{assocr\star} \,]\!]\ ((v_1, v_2), v_3) & = & (v_1, (v_2, v_3)) \\
[\![\, \mathsf{dist} \,]\!]\ (\mathsf{inl}\, v_1, v_3) & = & \mathsf{inl}\,(v_1, v_3) \\
[\![\, \mathsf{dist} \,]\!]\ (\mathsf{inr}\, v_2, v_3) & = & \mathsf{inr}\,(v_2, v_3) \\
[\![\, \mathsf{factor} \,]\!]\ (\mathsf{inl}\,(v_1, v_3)) & = & (\mathsf{inl}\, v_1, v_3) \\
[\![\, \mathsf{factor} \,]\!]\ (\mathsf{inr}\,(v_2, v_3)) & = & (\mathsf{inr}\, v_2, v_3) \\
[\![\, \mathsf{id}{\leftrightarrow_1} \,]\!]\ v & = & v
\end{array}
$$

$$
\begin{array}{lll}
[\![(c_1 \odot c_2)]\!]\ v & = & [\![\, c_2 \,]\!]([\![\, c_1 \,]\!]\ v) \\
[\![(c_1 \oplus c_2)]\!]\ (\mathsf{inl}\, v) & = & \mathsf{inl}\,([\![\, c_1 \,]\!]\ v) \\
[\![(c_1 \oplus c_2)]\!]\ (\mathsf{inr}\, v) & = & \mathsf{inr}\,([\![\, c_2 \,]\!]\ v) \\
[\![(c_1 \otimes c_2)]\!]\ (v_1, v_2) & = & ([\![\, c_1 \,]\!]\ v_1, [\![\, c_2 \,]\!]\ v_2)
\end{array}
$$

THEOREM 3.1. *The semantics is sound in the following sense:*

- *For every level-1 combinator $c : A \leftrightarrow_1 B$, we have a bijection $[\![\, c \,]\!]$ between $[\![\, A \,]\!]$ and $[\![\, B \,]\!]$.*
- *For every pair of combinators c_1 and c_2 of the same type $A \leftrightarrow_1 B$, if there exists a level-2 combinator $\alpha : c_1 \leftrightarrow_2 c_2$, then $[\![\, c_1 \,]\!] = [\![\, c_2 \,]\!]$, using extensional equivalence of functions.*

4 THE GROUPOID OF FINITE TYPES

In categorical language, the target for the semantics in the previous section is the category of finite sets and functions $\mathcal{S}\mathrm{et}_{\mathrm{fin}}$. However, as Π only refers to bijective functions, a more precise setting is the groupoid $\mathcal{B} = \mathrm{core}(\mathcal{S}\mathrm{et}_{\mathrm{fin}})$ of finite sets and bijections. $\mathcal{S}\mathrm{et}_{\mathrm{fin}}$ has finite coproducts $(\mathbf{0}, \sqcup)$ and finite products $(\mathbf{1}, \times)$, and in $\mathcal{B}$, these restrict to additive and multiplicative symmetric monoidal structures, respectively, making $\mathcal{B}$ a symmetric rig groupoid – the *vertical categorification* of the commutative rig of natural numbers $\mathbb{N}$ [Baez et al. 2010].

The semantics interprets types of Π as objects in $\mathcal{B}$, 1-combinators as isomorphisms, and for every pair of 1-combinators related by a 2-combinator, their interpretations in $\mathcal{B}$ are equal. $\mathcal{B}$ is a (strict) 1-groupoid where isomorphisms are identified upto extensional equality. We can evaluate two bijections on their inputs to decide if they are equal, and there is no higher cell witnessing the equality of two isomorphisms. To be able to establish completeness for Π, we want a witness for this equality, so that we can quote back to the syntax and produce a 2-combinator witnessing the equality of the corresponding 1-combinator.

We observe that the implicit equalities between the isomorphisms are pointwise equalities of functions, that is, homotopies. We therefore *weaken* the groupoid $\mathcal{B}$, exposing these homotopies, by using higher invertible cells. We work in HoTT/UF as it provides a proof-relevant, constructive metatheory to get a handle on these equalities and provides a rich internal language for describing weak groupoids, using the “types are weak ∞-groupoids” correspondence.

Every type in HoTT is a weak ∞-groupoid whose points are the terms of the type, and the (iterated) identity type gives the (higher) morphisms. The groupoid we are interested in has types as points, type equivalences for 1-cells, and higher homotopies for higher cells. (See Appendix B for an example on a 3-element set.) This is the groupoid structure for the universe type $\mathcal{U}$, since the identity type on types can be characterised as type equivalences (by *univalence*). But, we only want to carve out a *subuniverse* of *finite types*, still satisfying univalence, to get the groupoid structure. In this section, we formally define *univalent subuniverses*, and proceed to construct the particular instance for finite types, $\mathcal{U}_{\mathsf{Fin}}$ (see Definition 4.8).

4.1 The Type Theory

We use the type theory of the HoTT book [Univalent Foundations Program 2013], that is, we use intensional Martin-Löf Type Theory, with a hierarchy of (univalent) universes $\mathcal{U}_0 : \mathcal{U}_1 : \ldots$ (though we will just write $\mathcal{U}$ since we mostly use one universe), and a few Higher Inductive Types (HITs) for truncations and quotients. We write $\prod_{x:A} B(x)$ or simply $(x : A) \to B(x)$ for dependent function types, and $\sum_{x:A} B(x)$ or simply $(x : A) \times B(x)$ for dependent sum types. We use $\triangleq$ for definitions, $\equiv$ for definitional equalities, and $=$ for the identity type.

All arguments will hold in a Cubical Type Theory [Angiuli 2019; Cohen, Coquand, Huber, and Mörtberg 2018; Vezzosi, Mörtberg, and Abel 2019] as well. We review the basics of identity types, homotopy types, and some HITs that we use, and refer the reader to the book for more details.

4.2 Identity Types

Given two terms $x : A$ and $y : A$, we write $x =_A y$, or simply $x = y$, for the identity type, which is the type of equalities or identifications between them. The identity type is generated by reflexivity $\mathsf{refl}_x : x =_A x$, and the eliminator for the identity type is given by path induction or the J-rule (see Definition B.1 in the appendix). This construction can be iterated, giving the identity type between two terms of an identity type, repeating ad infinitum. Using the iterated identity type for morphisms, each type is equipped with the structure of a weak ∞-groupoid, where each morphism satisfies groupoid laws only upto a higher one.

A homotopy between functions $f, g : A \to B$, written $f \sim g$, is given by pointwise equality between them, $\prod_{x:A} f(x) =_B g(x)$. The identity type for functions is equivalent to homotopies between them $(f =_{A \to B} g) \simeq (f \sim g)$, by funext and happly. An equivalence between types $A \simeq B$ is given by a pair of functions between them which compose to the identity, $f \circ g \sim \mathsf{id}_B$ and $g \circ f \sim \mathsf{id}_A$ (also see Proposition 4.2), and this is equivalent to the identity type for the universe, $(A =_{\mathcal{U}} B) \simeq (A \simeq B)$, which is the univalence principle.

$$\mathsf{happly} : (f = g) \to (f \sim g) \qquad \mathsf{idtoeqv} : (A = B) \to (A \simeq B)$$
$$\mathsf{funext} : (f \sim g) \to (f = g) \qquad \mathsf{ua} : (A \simeq B) \to (A = B)$$

Functions between types are functors between groupoids. Given a function $f : A \to B$, there is a functorial action on the paths given by ap. Type families, that is, types indexed by terms, are simply functions from a type to the universe, such as $A \to \mathcal{U}$, which is an A-indexed family of groupoids. For a type family $P : A \to \mathcal{U}$ and a point $x : A$, the type $P(x)$ is the fiber over x. The transport operation lifts paths in the indexing type to functions between fibers.

$$\mathsf{ap}_f : \textstyle\prod_{x,y:A} x =_A y \to f(x) =_B f(y) \qquad \mathsf{transport}_P : \textstyle\prod_{x,y:A} x =_A y \to P(x) \to P(y)$$

The type $\sum_{x:A} P(x)$ is the collection of all the fibers and is the total space of P. The first projection $\pi_1 : \sum_{x:A} P(x) \to A$ from the total space to the base space A has the structure of a fibration, that is, there is a lifting operation (see Figure 12 in the appendix) which lifts paths in the base space to paths in the total space.

4.3 Univalent Fibrations

Using the groupoid structure of A, for any $x, y : A$ and a path $p : x =_A y$, $\mathsf{transport}(p)$ and $\mathsf{transport}(p^{-1})$ form an equivalence, lifting paths in the base space to equivalences between fibers.

$$\mathsf{transport\text{-}equiv}(P) : (x =_A y) \to (P(x) \simeq P(y))$$

The type families (or fibrations) we are interested in are the ones where paths in the base space completely determine the equivalences in the fibers – these are called univalent fibrations [Christensen 2015; Kapulkin, Lumsdaine, and Voevodsky 2012; Kapulkin and Lumsdaine 2021].

Definition 4.1 (Univalent Fibration). *P is a univalent type family (or, $\pi_1 : \sum_{x:A} P(x) \to A$ is a univalent fibration) if* $\mathsf{transport\text{-}equiv}(P)$ *is an equivalence.*

Univalent fibrations were introduced by Kapulkin, Lumsdaine, and Voevodsky [2012], to build a model of Voevodsky's *univalence* principle in simplicial sets. Indeed, univalence characterises paths in the universe as equivalences between types, which follows from the canonical fibration $\mathsf{id} : \mathcal{U} \to \mathcal{U}$ being univalent.

4.4 Higher Inductive Types

Higher Inductive Types generalise Inductive Types, by allowing path constructors besides point constructors. While point constructors generate the elements of the type, path constructors generate equalities between points in the type. We describe a few basic HITs that we use.

Given a type A, the propositional truncation $\|A\|_{-1}$, squashes the elements of A turning it into a proposition. It is given by a HIT with a point constructor $|{-}| : A \to \|A\|$, and a path constructor $\mathsf{trunc}(x, y) : x =_{\|A\|} y$, which equates every pair of points in the truncation (see Definition B.2).

Another HIT that we use is the set-quotient A/R which takes a set A and a binary relation $R : A \to A \to \mathcal{U}$. It has a mapping of points $\mathsf{q} : A \to A/R$, and a constructor that adds paths between related pairs of elements $\mathsf{q\text{-}rel} : R(x, y) \to \mathsf{q}(x) =_{A/R} \mathsf{q}(y)$ (see Definition B.3). We recall that if R is a prop-valued equivalence relation, then the quotient is *effective*, that is, $R(x, y)$ holds iff $(q(x) =_{A/R} q(y))$.

4.5 Homotopy Types

A type is *contractible* (a -2-type) if it has a unique element, that is, there is a center of contraction and every other point is equal to it. A type is a *proposition* (-1-type) if its equality types are contractible, that is, it has at most one inhabitant. Iterating this, we can define *sets* or 0-types (whose equality types are propositions) and *1-groupoids* or 1-types (whose equality types are sets), and similarly, higher homotopy n-types.

$$\mathsf{isContr}(A) \triangleq \textstyle\sum_{x:A} \prod_{y:A} y = x \qquad \mathsf{isSet}(A) \triangleq \prod_{x,y:A} \mathsf{isProp}(x = y)$$
$$\mathsf{isProp}(A) \triangleq \textstyle\prod_{x,y:A} \mathsf{isContr}(x = y) \qquad \mathsf{isGpd}(A) \triangleq \prod_{x,y:A} \mathsf{isSet}(x = y)$$

Given a function $f : A \to B$, there is an associated fibration over B, given by its homotopy fiber. The image of f is defined to be the (-1)-truncation of its fiber.

$$\mathsf{fib}_f(b) \triangleq \textstyle\sum_{a:A} f(a) =_B b \qquad \mathsf{im}(f) \triangleq \sum_{b:B} \left\|\mathsf{fib}_f(b)\right\|_{-1}$$

Finally, we recall a few characterisations of equivalences of homotopy types.

Proposition 4.2. *The following are equivalent.*

(1) $f : A \to B$ is an equivalence.
(2) f has a left and right inverse.
(3) f has contractible fibers.

4.6 Univalent Subuniverses

Starting from a univalent universe which classifies all types, we want to define a subuniverse which classifies only certain types, for example, types that satisfy some desired property. We use a prop-valued type family, that is, a predicate on the universe, which picks out only those types, and collect them into a univalent subuniverse. Being univalent ensures that the equality type of the ambient universe is reflected in the subuniverse.

Definition 4.3 (Universe). *A universe à la Tarski is given by the following pieces of data,*

- *a code* $U : \mathcal{U}$,
- *a decoding type family* $\mathsf{El} : U \to \mathcal{U}$.

If El *is univalent, we call* (U, El) *a* univalent *universe.*

Proposition 4.4 (Univalent Subuniverse). *A universe predicate is a type family* $P : \mathcal{U} \to \mathcal{U}$ *whose fibers are propositions, that is,* $P(X)$ *is a proposition for every* X*. Given such a predicate* P*, the fibration* $\pi_1 : \sum_{X:\mathcal{U}} P(X) \to \mathcal{U}$ *is univalent and generates a univalent subuniverse* $\mathcal{U}_P \triangleq (\sum_{X:\mathcal{U}} P(X), \pi_1)$.

The types we are interested in are the finite types. In constructive mathematics, the notion of finiteness is subtle [Spiwack and Coquand 2010]. We use the notion of Bishop-finiteness: a type is finite if it is merely equivalent to a finite set (Definitions 4.5 and 4.6).

Definition 4.5 (Fin). *The type family* $\mathsf{Fin} : \mathbb{N} \to \mathcal{U}$ *is the type of finite sets indexed by their cardinality. It is defined equivalently in two different ways,*

$$\mathsf{Fin}_n \triangleq \textstyle\sum_{k:\mathbb{N}} k < n \qquad \text{or} \qquad \begin{aligned} \mathsf{Fin}_0 &\triangleq \bot \\ \mathsf{Fin}_{S\,n} &\triangleq \top \sqcup \mathsf{Fin}_n \end{aligned}$$

Note that Fin_n is a set, and we use both definitions interchangeably.

Definition 4.6 (isFin). *We say that a type is finite if it is merely equal to* Fin_n *for some* $n : \mathbb{N}$.

$$\mathsf{isFin}(X) \triangleq \textstyle\sum_{n:\mathbb{N}} \|X =_{\mathcal{U}} \mathsf{Fin}_n\|_{-1}$$

Note that the natural number n need not be truncated, as justified below.

Lemma 4.7. *For any type* X, $\mathsf{isFin}(X)$ *is a proposition.*

Since isFin is a predicate on the universe $\mathcal{U}$, we easily get our univalent subuniverse $\mathcal{U}_{\mathsf{Fin}}$. This definition of the groupoid of finite types has also been considered in [Yorgey 2014].

Definition 4.8. *The univalent subuniverse of* all finite types *is given by* $\mathcal{U}_{\mathsf{Fin}} \triangleq \sum_{X:\mathcal{U}} \mathsf{isFin}(X)$. *We write* $F_n \triangleq (\mathsf{Fin}_n, n, |\,\mathsf{refl}\,|)$, *for the image of the inclusion of* Fin_n.

While $\mathcal{U}_{\mathsf{Fin}}$ has *all* the finite types, we are also interested in constructing a subuniverse of finite types of a *specified* cardinality. To do so, we will start with the subuniverse $\mathcal{B}\mathsf{Aut}(T)$, for any type $T : \mathcal{U}$. Characterisations of univalent fibrations using the $\mathcal{B}\mathsf{Aut}$ construction have also been studied by Christensen [2015].

Definition 4.9 ($\mathcal{B}$Aut). *The predicate* $P(X) \triangleq \|X \simeq T\|_{-1}$, *or equivalently,* $\|X = T\|_{-1}$, *picks out exactly those types that are merely equivalent to* T*, and this generates the subuniverse*

$$\mathcal{B}\mathsf{Aut}(T) \triangleq \textstyle\sum_{X:\mathcal{U}} \|X =_{\mathcal{U}} T\|_{-1}.$$

We write $T_0 \triangleq (T, |\,\mathsf{refl}_T\,|)$ *for the image of the inclusion of* T *in* $\mathcal{B}\mathsf{Aut}(T)$.

Using $\mathcal{B}$Aut, we can talk about types that are equivalent to a finite set of specified cardinality, for example, the subuniverse of 2-element sets is given by $\mathcal{B}\mathsf{Aut}(2)$. This has been used to construct the real projective spaces in HoTT [Buchholtz and Rijke 2017], and also to give sound and complete denotational semantics to a 1-bit fragment of Π [Carette, Chen, Choudhury, and Sabry 2018].

Definition 4.10 ($\mathcal{U}_{\mathsf{Fin}_n}$). *For any* $n : \mathbb{N}$, *we define* $\mathcal{U}_{\mathsf{Fin}_n} \triangleq \mathcal{B}\mathsf{Aut}(\mathsf{Fin}_n)$ *to be the univalent subuniverse of* n*-element sets. Note that,* $\mathcal{U}_{\mathsf{Fin}}$ *can be equivalently seen as the collection of all finite types, that is,* $\mathcal{U}_{\mathsf{Fin}} \simeq \sum_{n:\mathbb{N}} \mathcal{U}_{\mathsf{Fin}_n}$.

Since $\mathcal{B}\mathsf{Aut}(T)$ is a univalent subuniverse, we can characterise its path space. The intuition is that $\mathcal{B}\mathsf{Aut}(T)$ only has one point or 0-cell T_0, and 1-paths $T_0 = T_0$, that is, loops, and higher paths between these loops. The type of loops on T_0, that is, $\Omega(\mathcal{B}\mathsf{Aut}(T), T_0)$, is shown to be equivalent to $\mathsf{Aut}(T) \triangleq T \simeq T$, which is the group of automorphisms of T.

Lemma 4.11.

(1) If T is an n-type, $\mathcal{B}\mathsf{Aut}(T)$ is an $(n+1)$-type.
(2) For any $T : \mathcal{U}$, $\mathcal{B}\mathsf{Aut}(T)$ is 0-connected.
(3) For any $T : \mathcal{U}$, $\Omega(\mathcal{B}\mathsf{Aut}(T), T_0) \simeq \mathsf{Aut}(T)$.

Theorem 4.12.

(1) $\mathcal{U}_{\mathsf{Fin}_n}$ is a pointed, connected, 1-groupoid for every $n : \mathbb{N}$, and $\Omega(\mathcal{U}_{\mathsf{Fin}_n}, F_n) \simeq \mathsf{Aut}(\mathsf{Fin}_n)$.
(2) $\mathcal{U}_{\mathsf{Fin}}$ is a 1-groupoid with connected components for every $n : \mathbb{N}$.

We have shown that loops in $\mathcal{U}_{\mathsf{Fin}}$ exactly encode the automorphism group $\mathsf{Aut}(\mathsf{Fin}_n)$ for every n. This is a general technique called *delooping*, where a group can be identified with a 1-object groupoid, that is, a 0-connected type in HoTT. This technique also allows defining higher groups [Buchholtz et al. 2018]. The loopspace of a pointed type automatically has the structure of a group, with $\mathsf{refl}_\star$ for the neutral element, path composition for the group multiplication, and path inverse for the group inverse. The group axioms follow from the higher paths corresponding to groupoid laws.

4.7 Rig Structure

Similar to $\mathbb{B}$, the groupoid $\mathcal{U}_{\mathsf{Fin}}$ has two symmetric monoidal structures, the additive and the multiplicative ones, and the multiplicative tensor product distributes over the additive one. To construct these, we first state and prove some equivalences on Fin, and some general type isomorphisms. Then we simply lift these equivalences to $\mathcal{U}_{\mathsf{Fin}}$, by the univalence principle.

Proposition 4.13. *For any $n, m : \mathbb{N}$, and for any types X, Y, Z,*

$$\begin{array}{cc}
\mathsf{Fin}_0 \simeq \bot & \mathsf{Fin}_1 \simeq \top \\
\mathsf{Fin}_n \sqcup \mathsf{Fin}_m \simeq \mathsf{Fin}_{n+m} & \mathsf{Fin}_n \times \mathsf{Fin}_m \simeq \mathsf{Fin}_{n \cdot m} \\
\\
\bot \sqcup X \simeq X & \top \times X \simeq X \\
X \sqcup \bot \simeq X & X \times \top \simeq X \\
(X \sqcup Y) \sqcup Z \simeq X \sqcup (Y \sqcup Z) & (X \times Y) \times Z \simeq X \times (Y \times Z) \\
X \sqcup Y \simeq Y \sqcup X & X \times Y \simeq Y \times X \\
X \times \bot \simeq \bot & X \times (Y \sqcup Z) \simeq (X \times Y) \sqcup (X \times Z)
\end{array}$$

Theorem 4.14. *$\mathcal{U}_{\mathsf{Fin}}$ has two symmetric monoidal structures, the additive and multiplicative ones, given by $(F_0, \sqcup)$ and $(F_1, \times)$, with corresponding natural isomorphisms λ_X, ρ_X, $\alpha_{X,Y,Z}$, and the braiding isomorphism $\mathcal{B}_{X,Y}$, upto 1-paths in $\mathcal{U}_{\mathsf{Fin}}$. These isomorphisms satisfy the Mac Lane coherence conditions for symmetric monoidal categories [MacLane 1963], that is, the triangle, pentagon, and hexagon identities, and the symmetry of the braiding, upto 2-paths in $\mathcal{U}_{\mathsf{Fin}}$. The multiplicative structure distributes over the additive structure and satisfies the Laplaza coherence conditions for rig categories [Laplaza 1972].*

5 THE GROUP OF PERMUTATIONS

In Section 4, we established that paths in $\mathcal{U}_{\mathsf{Fin}}$ are equivalent to families of loops on Fin_n for every $n : \mathbb{N}$, that is, automorphisms of finite sets of size n, with the loopspace encoding the automorphism group. This is also the finite symmetric group S_n, making $\mathcal{U}_{\mathsf{Fin}}$ the *horizontal categorification* of S_n for every n. In this section, we will describe this group syntactically.

In order to study syntactic descriptions of permutations, we will hit the problem of deciding whether two descriptions refer to the same permutation – in group theory, this is the *word problem* for S_n. Putting it in this form allows us to connect it to the broader scope of computational group theory and combinatorics – we can borrow ideas such as Coxeter relations and Lehmer codes. The concepts we use can be found in any standard textbook on group theory, or see the Symmetry book [Bezem et al. 2021] for a univalent point of view.

Thus, the goal of this section is to reconcile two different approaches to defining the symmetric group – as an automorphism group, and as a group syntactically presented using *generators* and *relations* [Matsumoto 1964]. The generators of the group are similar to the primitive combinators in a (reversible) programming language – the group structure gives the composition and inverse operations, and the relations describe how these primitive combinators interact with each other.

First, we will define the required notions of free groups and group presentations, and state some of their most important properties. Then, we introduce our chosen *Coxeter presentation* for S_n. To solve the word problem for S_n, we will use a rewriting system, with a suitable, well behaved collection of reduction rules corresponding to the Coxeter presentation equations. Finally, we describe the normal forms in this rewriting system, using Lehmer codes, and prove the correspondence between them and the type $\mathsf{Aut}(\mathsf{Fin}_\mathsf{n})$ of automorphisms on a finite set. The generators and relations we use here will be used to quote back to 1 and 2-combinators in Π (see Section 6).

5.1 Presenting the Symmetric Group

One way of thinking about presentations of S_n is via sorting algorithms, which use different primitive operations. A sorting algorithm has to calculate a permutation of a list or a (listed) finite set, which satisfies the invariant of being a sorted sequence, which means, the primitive operations of a sorting algorithm are able to generate all the permutations on a given list. So, a chosen set of reversible operations in a sorting algorithm can be a good candidate for the generators of symmetric groups. For example, we could generate the symmetric group on Fin_n by using generators (primitive operations) that:

- swap the i-th element with the $(i + 1)$-th element, that is, adjacent swaps, or
- swap the i-th element with the j-th element, for arbitrary i-s and j-s, or
- swap the i-th element with an element at a fixed position, or
- reverse a prefix Fin_k of Fin_n for $k \leq n$, or
- cyclically shift any subset of Fin_n.

Bubble sort uses the primitive operation of adjacent swaps, insertion sort and selection sort use the primitive operation of swapping the i-th element with the j-th element, cycle sort uses cyclical shifts of subsequences, pancake sort uses reversals of the prefixes of the list, et cetera. The choice of generators for our presentation is important for the following reasons.

- It affects the difficulty of solving the word problem in S_n and formalising the proof of its correctness.
- The choice of generators dictates which words become normal forms in this presentation of S_n. These normal forms dictate the shape of the synthesised and normalised boolean circuits, which is the application we have in mind.
- Finally, the generators have to closely match the Π combinators so that we can quote back a permutation to a Π program, for the proof of completeness.

We will show in Section 6 that all Π combinators can be encoded using adjacent transpositions.

5.2 Free Groups

Usually, there are many equations, besides the group axioms, that hold for the elements of a group. For example, in the group $\mathsf{Aut}(\mathbf{2})$, or $\mathbb{Z}_2$, we have an equation $1 + 1 = 0$, which is not a consequence of the group axioms, but is specific to this particular group. A free group has the property that no other equations hold except the ones directly implied by the group axioms. For example, the additive group of integers $\mathbb{Z}$ is the free group on the singleton set.

A group homomorphism between groups G and H is a function $f : G \to H$ between the underlying sets that preserves the group structure. Giving a group homomorphism out of the free group is equivalent to giving a function out of the generating set. This is the universal property of free groups, stemming from the free-forgetful adjunction between the categories of groups and sets. We say that $F(A)$ with $\eta_A : A \to F(A)$ is the free group on A if it satisfies the following.

Proposition 5.1 (Universal Property of $F(A)$). *Given a group G and a map $f : A \to G$, there is a unique group homomorphism $f^\# : \mathsf{Hom}(F(A), G)$ such that $f^\# \circ \eta_A \sim f$. Equivalently, composition with η_A gives an equivalence $\mathsf{Hom}(F(A), G) \simeq A \to G$.*

Following the naive universal-algebraic definition, in HoTT, we could use a higher inductive type to define the free group, which enforces the group axioms by adding path constructors (see Definition C.2). Using the induction principle, we can easily verify the universal property. However, since this definition of $F(A)$ has lots of path constructors corresponding to each group axiom, characterising its path space is difficult.

Instead, we will think about elements of the free group as words over an alphabet of letters drawn from the generating set *and* the set of their formal inverses. If we take the disjoint union of A with itself, that is, $A + A$ as the group's underlying set, we can use $\mathsf{inl}/\mathsf{inr}$ to mark the elements – $\mathsf{inl}\, a$ means a and $\mathsf{inr}\, a$ means a^{-1}. Then, we can encode the free group using the free monoid, that is, lists of $A + A$. Additionally, we need to ensure that the inverse laws hold, so we have to coalesce adjacent occurrences of a and a^{-1}.

Definition 5.2 (Free group). *Let A be a set, and $\mathsf{List}(-)$ the free monoid monad. The free group $F(A)$ on A is the set-quotient of $\mathsf{List}(A + A)$ by the congruence closure of the relation $a :: a^{-1} :: \mathsf{nil} \sim \mathsf{nil}$ and $a^{-1} :: a :: \mathsf{nil} \sim \mathsf{nil}$.*

Proposition 5.3. *$F(A) \triangleq \mathsf{List}(A + A)/{\sim^*}$ has a group structure, with the empty list nil for the neutral element, multiplication given by list append $+\!\!+$, and inverse given by flipping inl and inr, followed by reversing the list. Further, $F(A)$ with $\eta_A(a) \triangleq \mathsf{inl}(a) :: \mathsf{nil}$ satisfies the universal property of free groups, as stated in Proposition 5.1.*

5.3 Group Presentations

A presentation of a group builds it by starting from the free group $F(A)$ and introducing additional equations that are satisfied in the resulting group. For example, if we take $F(\mathbf{1}) \triangleq \mathbb{Z}$ and add an equation $1 + 1 = 0$, the resulting group would be $\mathbb{Z}_2 \simeq \mathsf{Aut}(\mathbf{2})$. Note that not all groups have finite (or computable) presentations, and, a group can have a number of different presentations.

Definition 5.4 (Group presentation). *Let A be a set and $R : \mathsf{List}(A + A) \to \mathsf{List}(A + A) \to \mathcal{U}$ a binary relation on $\mathsf{List}(A + A)$. The group $F(\langle A; R\rangle)$ presented by A and R, is given by the set-quotient of the free group $F(A)$ by the congruence closure of R.*

The universal property of the above definition of a presented group is similar to Proposition 5.1, except the function mapping out of the generating set also has to respect the relation.

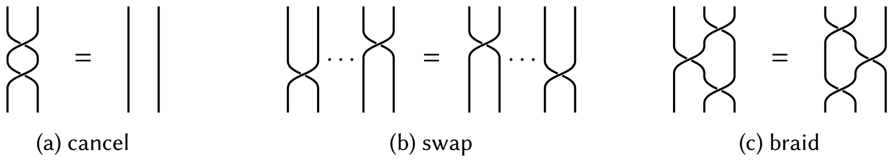


Fig. 5. Braiding diagrams for Coxeter relations.

Proposition 5.5 (Universal property of $F(\langle A; R\rangle)$). *Given a group G and a map $f : A \to G$, such that f extended to $F(A)$ respects R, there is a unique group homomorphism $f^{\#} : \mathsf{Hom}(F(\langle A; R\rangle), G)$ such that $f^{\#} \circ \eta_A \sim f$.*

Before, the only way to decide the equality of two elements in a group was to evaluate and check them on the nose, but in a group presentation, this is reduced to deciding whether one word – a representative of the equivalence class of the group's elements, can be reduced to another word, using the group's relations. However, these relations are not directed, so it is not always possible to construct a well-behaved rewriting system. In general, the word problem for groups is undecidable.

Coxeter Presentation. To present the group $\mathsf{S_n}$, the primitive operations we use will be adjacent swaps. When dealing with permutations on an $n+1$-element set, there are n adjacent transpositions – transposition number k swaps elements at indices k and $k+1$. Thus, the generating set is $\mathsf{Fin_n}$. There are three relations that we're going to specify for this presentation – we visualise them as braiding diagrams in Figure 5.

5a Swapping the same two elements twice in a row should be the same as doing nothing.
5b When swapping two distinct pairs of elements, the order in which swapping happens should not matter, that is, we can slide the wires freely.
5c There are two equivalent ways of swapping the first and last elements in a sequence of three elements.

This construction is called a Coxeter presentation of $\mathsf{S_n}$. Writing it formally, we encode the rules discussed above using a relation $\leftrightsquigarrow$ on $\mathsf{List}(\mathsf{Fin_n})$ (Definition 5.6), and then take its congruence closure $\overset{*}{\leftrightsquigarrow}$ (Definition C.3).

Definition 5.6 ($\leftrightsquigarrow$: $\mathsf{List}(\mathsf{Fin_n}) \to \mathsf{List}(\mathsf{Fin_n}) \to \mathcal{U}$).

$$\begin{aligned}
\mathsf{cancel} &: \forall n \to (n :: n :: \mathsf{nil}) \leftrightsquigarrow \mathsf{nil} \\
\mathsf{swap} &: \forall k, n \to (\mathsf{S}\,k < n) \to (n :: k :: \mathsf{nil}) \leftrightsquigarrow (k :: n :: \mathsf{nil}) \\
\mathsf{braid} &: \forall n \to (\mathsf{S}\,n :: n :: \mathsf{S}\,n :: \mathsf{nil}) \leftrightsquigarrow (n :: \mathsf{S}\,n :: n :: \mathsf{nil})
\end{aligned}$$

The idea for solving the word problem for $\mathsf{S_n}$ is to turn these undirected relations into a rewriting system $(\mathsf{List}(\mathsf{Fin_n}), \overset{*}{\leftrightsquigarrow})$, so that, by repeatedly applying the reduction rules as long as possible, any two $\overset{*}{\leftrightsquigarrow}$-equal terms would eventually converge to the same normal form.

For this to work, we first need the system to have the termination property, meaning that there should be no infinite reductions. We observe that, after throwing out reflexivity and symmetry, the right-hand sides of the relations $\overset{*}{\leftrightsquigarrow}$ are strictly smaller than the left-hand sides, in terms of the lexicographical ordering on words in $\mathsf{Fin_n}$ (which is well-founded). Thus, by directing the relation from left to right, we would get the termination property out of the box. Second, we need the normal forms to be unique, so that we can extract a normalisation function. This will be true if the rewriting system is confluent – all critical pairs, that is, terms with overlapping possible reduction rules, have to converge. For example, in our system, the pair in Figure 6a converges. Unfortunately,

$$\begin{tikzcd}[column sep=small]
& \overline{\tau_2\tau_1\underline{\tau_2}}\underline{\tau_1\tau_2} \arrow[dl, squiggly, "\text{braid}"'] \arrow[dr, squiggly, "\text{braid}"] & \\
\tau_1\tau_2\overline{\tau_1\tau_1}\tau_2 \arrow[d, squiggly, "\text{cancel}"'] & & \tau_2\underline{\tau_1\tau_1}\tau_2\tau_1 \arrow[d, squiggly, "\text{cancel}"] \\
\tau_1\overline{\tau_2\tau_2} \arrow[dr, squiggly, "\text{cancel}"'] & & \underline{\tau_2\tau_2}\tau_1 \arrow[dl, squiggly, "\text{cancel}"] \\
& \tau_1 &
\end{tikzcd}$$

(a) Converging critical pair

$$\begin{tikzcd}[column sep=small]
& \overline{\tau_3\tau_2\underline{\tau_3}}\underline{\tau_1} \arrow[dl, squiggly, "\text{braid}"'] \arrow[dr, squiggly, "\text{swap}"] & \\
\tau_2\tau_3\tau_2\tau_1 & & \underline{\tau_3\tau_2\tau_1\tau_3} \arrow[ll, squiggly, "\text{braid}^\blacklozenge"]
\end{tikzcd}$$

(b) Non-converging critical pair that converges with braid$^\blacklozenge$

Fig. 6. Critical Pairs in $\overset{*}{\leftrightsquigarrow}$

this is not true for all critical pairs – an example is in Figure 6b, where left and right endpoints are normal with respect to the $\overset{*}{\leftrightsquigarrow}$ relation.

5.4 Rewriting via Coxeter

Because of this counter-example, the relations have to be changed. In this section, we formally define a rewriting system $(\mathsf{List}(\mathsf{Fin}_n), \rightsquigarrow)$, partially based on the Coxeter relations, and prove that it has the desired properties of confluence and termination. We prove that the new relation defined by this system is equivalent, in a technical sense (Proposition 5.10), to the standard Coxeter relation $\overset{*}{\leftrightsquigarrow}$. First, we need to define a function $\swarrow$.

Definition 5.7 ($\swarrow : (n : \mathbb{N}) \to (k : \mathbb{N}) \to \mathsf{List}(\mathsf{Fin}_{k+n})$).

$$n \swarrow 0 \triangleq \mathsf{nil}$$
$$n \swarrow \mathsf{S}\,k \triangleq (k+n) :: (n \swarrow k)$$

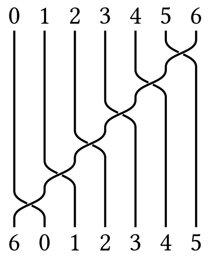
The result of computing $n \swarrow k$ is the sequence $[k+n-1, k+n-2, k+n-3, \ldots, n]$, which is a sequence of transpositions that moves the element at index $k+n$ left by k places, shifting all the elements in between one place right (see Figure 7a). Then, the directed relation $\rightsquigarrow$ is defined with the following generators (inlining the congruence closure in the relation $\rightsquigarrow$ allows arbitrary reductions inside the list).

Definition 5.8 ($\rightsquigarrow : \mathsf{List}(\mathsf{Fin}_n) \to \mathsf{List}(\mathsf{Fin}_n) \to \mathcal{U}$).

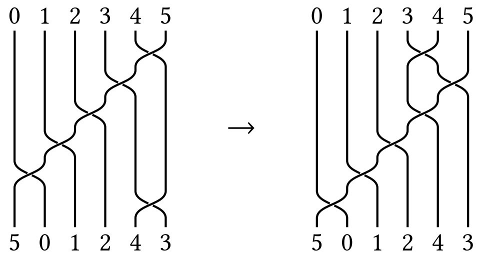
$$\mathsf{cancel}^\blacklozenge : \forall n, l, r \to (l +\!\!+ n :: n :: r) \rightsquigarrow (l +\!\!+ r)$$
$$\mathsf{swap}^\blacklozenge : \forall n, k, l, r \to (\mathsf{S}\,k < n) \to (l +\!\!+ n :: k :: r) \rightsquigarrow (l +\!\!+ k :: n +\!\!+ r)$$
$$\mathsf{braid}^\blacklozenge : \forall n, k, l, r \to (l +\!\!+ (n \swarrow 2+k) +\!\!+ (1+k+n) :: r) \rightsquigarrow (l +\!\!+ (k+n) :: (n \swarrow 2+k) +\!\!+ r)$$

Constructors cancel$^\blacklozenge$ and swap$^\blacklozenge$ correspond directly to the appropriate constructors of $\leftrightsquigarrow$ and can be visualised in the same way as before. The remaining constructor braid$^\blacklozenge$ uses the $\swarrow$ function to exchange the order of a long sequence of transpositions and a single transposition afterwards. For example, for $n = 0$ and $k = 3$, it allows for the reduction $[4,3,2,1,0,4] \rightsquigarrow [3,4,3,2,1,0]$ (see Figure 7b – there is a distinction between wires, where numbers represent the values, and transpositions, where numbers represent crossing wires).

Note that the previous braid rule is a special case of braid$^\blacklozenge$, with $k = 0$. As before, the left-hand sides of the relation are (lexicographically) strictly larger than the right-hand sides. We define the



(a) $0 \swarrow 6$



(b) braid$^\blacklozenge$ with $n = 0$ and $k = 3$

Fig. 7. Braiding diagrams for modified Coxeter relations.

transitive closure of $\rightsquigarrow$ to be $\overset{+}{\rightsquigarrow}$ (see Definition C.5 in the appendix) and its reflexive-transitive closure to be $\overset{*}{\rightsquigarrow}$ (see Definition C.4).

Despite the increased complexity of the generators, the rewriting system $(\mathsf{List}(\mathsf{Fin}_n), \rightsquigarrow)$ has the properties we desire. It satisfies local confluence, that is, the Church-Rosser (diamond) property – for example, using braid$^\blacklozenge$, we can now show the problematic critical pair Figure 6b converges, and, it is terminating, so by Newman's lemma, it produces a unique normal form. We follow the terminology of Huet [1980]; Kraus and von Raumer [2020] to state our results formally.

Theorem 5.9.

(1) $\rightsquigarrow$ is (locally) confluent. For every span $w_2 \leftsquigarrow w_1 \rightsquigarrow w_3$, there is a matching extended cospan $w_2 \overset{*}{\rightsquigarrow} w \overset{*}{\leftsquigarrow} w_3$.

(2) $\overset{+}{\rightsquigarrow}$ is terminating. For every $w \overset{+}{\rightsquigarrow} v$, it holds that $v < w$, where $<$ is the (well-founded) lexicographic ordering on $\mathsf{List}(\mathsf{Fin}_n)$.

(3) $\overset{*}{\rightsquigarrow}$ is confluent. For every extended span $w_2 \overset{*}{\leftsquigarrow} w_1 \overset{*}{\rightsquigarrow} w_3$, there is a matching extended cospan $w_2 \overset{*}{\rightsquigarrow} w \overset{*}{\leftsquigarrow} w_3$.

(4) For every w, there exists a unique normal form v such that $w \overset{*}{\rightsquigarrow} v$.

The modified form of the Coxeter relations are unwieldy and difficult to prove properties about by induction. However, we can make the following observation relating it to $\overset{*}{\leftrightsquigarrow}$.

Proposition 5.10. *The relations $\overset{*}{\leftrightsquigarrow}$ and $\overset{*}{\rightsquigarrow}$ are equivalent in the following sense: for every w and v, $w \overset{*}{\leftrightsquigarrow} v$ iff there is a u such that $w \overset{*}{\rightsquigarrow} u \overset{*}{\leftsquigarrow} v$.*

By Theorem 5.9 part 4, we get a unique choice function $\mathfrak{nf} : \mathsf{List}(\mathsf{Fin}_n) \longrightarrow \mathsf{List}(\mathsf{Fin}_n)$ that produces a normal form for terms of $\mathsf{List}(\mathsf{Fin}_n)$. We state two important properties enjoyed by $\mathfrak{nf}$.

Proposition 5.11.

(1) For all $l : \mathsf{List}(\mathsf{Fin}_n)$, we have that $l \overset{*}{\leftrightsquigarrow} \mathfrak{nf}(l)$.

(2) $\mathfrak{nf}$ is idempotent, that is, $\mathfrak{nf} \circ \mathfrak{nf} \sim \mathfrak{nf}$.

Finally, we define the type S_n as the set-quotient of $\mathsf{List}(\mathsf{Fin}_n)$ by $\overset{*}{\leftrightsquigarrow}$,

Definition 5.12 (S_n). $\mathsf{S}_n \triangleq \mathsf{List}(\mathsf{Fin}_n)/\overset{*}{\leftrightsquigarrow}$

Note that reductions need not be unique, hence $w \overset{*}{\leftrightsquigarrow} v$ is not necessarily a proposition. So, the quotient S_n is not effective, that is, $\mathsf{q\text{-}rel} : w \overset{*}{\leftrightsquigarrow} v \longrightarrow q(w) = q(v)$ is not an equivalence. Using the $\mathfrak{nf}$ function, we could instead define a new relation $\approx$ to equate those terms that have the same normal form, $(w \approx v) \triangleq (\mathfrak{nf}(w) = \mathfrak{nf}(v))$. This relation is prop-valued, and we could quotient $\mathsf{List}(\mathsf{Fin_n})$ by $\approx$, obtaining an equivalent definition for $\mathsf{S_n}$.

Proposition 5.13.

(1) $\mathfrak{nf}$ *splits into a section-retraction pair, that is, we have* $\mathsf{List}(\mathsf{Fin_n}) \xrightarrow{s} \mathsf{S_n} \xrightarrow{r} \mathsf{List}(\mathsf{Fin_n})$ *such that* $s \circ r \sim \mathfrak{nf}$ *and* $r \circ s \sim \mathsf{id}_{\mathsf{S_n}}$.

(2) $\mathsf{im}(\mathsf{q}) \simeq \mathsf{S_n} \simeq \mathsf{im}(\mathfrak{nf})$, *where* $\mathsf{q} : \mathsf{List}(\mathsf{Fin_n}) \to \mathsf{S_n}$ *is the mapping into the quotient.*

Finally, we need to show that $\mathsf{S_n}$ is indeed a group presentation.

Proposition 5.14. *There is a group structure on* $\mathsf{S_n}$*, where the identity element is* nil*, multiplication is given by list append, and inverse is given by list reversal.*

Notice however, that for a group presentation, as defined in Definition 5.4, the relation needs to be on the set of words $A + A$, where the right copy corresponds to the set of formal inverses of the generators. The following proposition lets us lift a group structure on $\mathsf{List}(A)$ to a group presentation, whose underlying set is $\mathsf{List}(A + A)$.

Proposition 5.15. *Given a set A and a congruence relation R on* $\mathsf{List}(A)$ *with the group structure given by append and reversal, there is a generated group structure $F(\langle A; R_+ \rangle)$, where R_+ is the relation R lifted to* $\mathsf{List}(A + A)$ *along the codiagonal map $\nabla_A : A + A \to A$.*

Corollary 5.16. *The group* $\mathsf{S_n}$ *is isomorphic to the group presentation given by* $F(\langle \mathsf{Fin_n}; \overset{*}{\leftrightsquigarrow}_+ \rangle)$.

To decide if two words in $\mathsf{List}(\mathsf{Fin_n})$ are $\overset{*}{\leftrightsquigarrow}$-equal, we simply have to compute their normal forms using $\mathfrak{nf}$. They correspond to the same permutation if and only if these normal forms are equal, which is decidable for $\mathsf{List}(\mathsf{Fin_n})$.

5.5 Lehmer Codes

To prove the equivalence between $\mathsf{Aut}(\mathsf{Fin_n})$ and $\mathsf{S_n}$, we will define functions back and forth between the two types. The terms in $\mathsf{S_n}$ can be identified with equivalence classes of terms in $\mathsf{List}(\mathsf{Fin_n})$ with respect to the Coxeter relation $\overset{*}{\leftrightsquigarrow}$. The easiest way to define a function out of this presentation is to define it on the representatives. We know that these are the unique normal forms in the set-quotient given by $\mathsf{q} \circ \mathfrak{nf}$, but now we will explicitly describe what these representatives look like, using an encoding called Lehmer codes [Lehmer 1960].

Lehmer codes are known in Combinatorial Analysis [Bellman and Hall 1960] where they are sometimes called "subexcedant sequences", or "factoriadics". They can be written as a decimal number in the factorial number system, or as a tuple encoding the digits [Knuth 1997; Laisant 1888]. This gives a convenient way of representing permutations on a computer, partly because they are bitwise-optimal [Berger et al. 2019].

Formally, we define $\mathsf{Lehmer}(n)$ to be an $n + 1$-element tuple, where the position $k \leq n$ stores an element of $\mathsf{Fin_k}$. Since the 0-th position is trivial, in practice it is ignored [Dubois and Giorgetti 2018; Vajnovszki 2011].

Definition 5.17 ($\mathsf{Lehmer} : \mathbb{N} \to \mathcal{U}$).

$$\mathsf{Lehmer}(0) \triangleq \mathsf{Fin}_{\mathsf{S}\,0}$$
$$\mathsf{Lehmer}(\mathsf{S}\,n) \triangleq \mathsf{Lehmer}(n) \times \mathsf{Fin}_{\mathsf{S}\,\mathsf{S}\,n}$$

The inversion count of an element is given by the number of smaller elements that appear after it in the permutation.

Definition 5.18 (Inversion count). *Given a permutation* $\sigma : \mathsf{Aut}(\mathsf{Fin}_n)$, *the inversion count of an element* $i : \mathsf{Fin}_n$ *is given by*

$$\mathsf{inv}_\sigma(i) \triangleq \#\{\, j < i \mid \sigma(j) > \sigma(i) \,\}\,.$$

Knowing the inversion counts for all the elements, one can reconstruct the starting permutation. Also, observe that $\mathsf{inv}_\sigma(i) < i$, thus fitting in the i-th place of a Lehmer code tuple. As an example, consider the following tabulated presentation of a permutation of Fin_5.

$$\sigma \triangleq \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ \mathbf{2} & \mathbf{1} & \mathbf{4} & \mathbf{0} & \mathbf{3} \end{pmatrix}$$

The Lehmer code for the permutation σ is then the 5-tuple

$$l \equiv (\mathsf{inv}_\sigma(0), \mathsf{inv}_\sigma(1), \mathsf{inv}_\sigma(2), \mathsf{inv}_\sigma(3), \mathsf{inv}_\sigma(4)) \equiv (0, 1, 2, 0, 2)$$

To decode the permutation back from this Lehmer code, we perform an algorithm similar to *insertion sort*. The element of the Lehmer code being currently processed is highlighted in the left column of the table below. Starting from a sorted list, the element at index k has to be given $l[k]$ inversions. Because of the invariant that all the elements before newly processed one are smaller than it, the proper number of inversions is created by simply shifting the element $l[k]$ places left.

(0, 1, 2, 0, 2)	$[0, \mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}]$
(0, 1, 2, 0, 2)	$[1, \mathbf{0}, \mathbf{2}, \mathbf{3}, \mathbf{4}]$
(0, 1, 2, 0, 2)	$[2, \mathbf{1}, \mathbf{0}, \mathbf{3}, \mathbf{4}]$
(0, 1, 2, 0, 2)	$[\mathbf{2}, \mathbf{1}, \mathbf{0}, 3, \mathbf{4}]$
(0, 1, 2, 0, 2)	$[\mathbf{2}, \mathbf{1}, 4, \mathbf{0}, \mathbf{3}]$

Writing formally, to turn a Lehmer code into a word in S_n, we define a function $\mathfrak{em}$. As described above, the number r at position k in the tuple describes how many inversions the element $\mathbf{k}$ has. Thus, we need to perform r many adjacent transpositions to get to the desired position, which is given by $(\mathsf{S}\,n - r) \swarrow r$.

Definition 5.19 ($\mathfrak{em} : (n : \mathbb{N}) \longrightarrow \mathsf{Lehmer}(n) \longrightarrow \mathsf{List}(\mathsf{Fin}_{\mathsf{S}\,n})$).

$$\mathfrak{em}_0(0) \triangleq \mathsf{nil}$$

$$\mathfrak{em}_{\mathsf{S}\,n}((r, l)) \triangleq \mathfrak{em}_n(l) +\!\!+ ((\mathsf{S}\,n - r) \swarrow r)$$

We can show that the function $\mathfrak{em}_n$ gives an equivalence between $\mathsf{Lehmer}(n)$ and $\mathsf{im}(\mathfrak{nf})$.

Theorem 5.20.

(1) For any Lehmer code c, $\mathfrak{em}_n(c)$ *is a normal form with respect to* $\overset{*}{\rightsquigarrow}$, *that is,* $\mathfrak{em}_n(c)$ *is in* $\mathsf{im}(\mathfrak{nf})$.

(2) Any element of $\mathsf{im}(\mathfrak{nf})$ *can be constructed from a unique Lehmer code by* $\mathfrak{em}$, *that is, the fibers of* $\mathfrak{em}_n : \mathsf{Lehmer}(n) \longrightarrow \mathsf{im}(\mathfrak{nf})$ *are contractible.*

Therefore, there is an equivalence between $\mathsf{Lehmer}(n)$ *and* $\mathsf{im}(\mathfrak{nf})$ *(see Proposition 4.2).*

Corollary 5.21. *For all* $n : \mathbb{N}$, $\mathsf{S}_n \simeq \mathsf{im}(\mathfrak{nf}) \simeq \mathsf{Lehmer}(n)$.

5.6 Running Lehmer Codes

Finally, it is time to complete our goal of characterising the symmetric group. Having produced a Lehmer code by normalising words in S_n, we need to run it to produce a concrete bijection of finite sets, and, given a bijection between finite sets, we need to encode it as a Lehmer code. We will prove that these maps construct an equivalence between the types $\mathsf{Lehmer}(n)$ and $\mathsf{Aut}(\mathsf{Fin}_{\mathsf{S}\,n})$ [1]. The idea for this proof is borrowed from Molzer [2021].

[1]Note that the indices for the type of permutations are off-by-one, because we chose Fin_n to represent generators for permutations on $\mathsf{Fin}_{\mathsf{S}\,n}$.

Definition 5.22 ($\mathsf{Fin}^{-}_{\mathsf{n}} : \mathsf{Fin}_{\mathsf{n}} \to \mathcal{U}$). *For $n : \mathbb{N}$, the type family $\mathsf{Fin}^{-}_{\mathsf{n}}$ picks out all elements in $\mathsf{Fin}_{\mathsf{n}}$ except the one in the argument.*

$$\mathsf{Fin}^{-}_{\mathsf{n}}(i) \triangleq \textstyle\sum_{j:\mathsf{Fin}_{\mathsf{n}}} i \neq j$$

Note that $\mathsf{Fin}^{-}_{\mathsf{n}}(i)$ for $i : \mathsf{Fin}_{\mathsf{n}}$ is a subtype of $\mathsf{Fin}_{\mathsf{n}}$ and is hence a set. We state and prove a few auxiliary lemmas about how $\mathsf{Fin}^{-}_{\mathsf{n}}$ interacts with Fin – these are obtained by counting arguments using the fact that $\mathsf{Fin}_{\mathsf{n}}$ and $\mathsf{Fin}^{-}_{\mathsf{n}}$ both have decidable equality.

Lemma 5.23.

(1) For any $k : \mathsf{Fin}_{\mathsf{S\,n}}$, we have $\mathsf{Fin}^{-}_{\mathsf{S\,n}}(k) \simeq \mathsf{Fin}_{\mathsf{n}}$.
(2) For any $n : \mathbb{N}$, we have $\mathsf{Aut}(\mathsf{Fin}_{\mathsf{S\,n}}) \simeq \sum_{k:\mathsf{Fin}_{\mathsf{S\,n}}} (\mathsf{Fin}^{-}_{\mathsf{S\,n}}(n) \simeq \mathsf{Fin}^{-}_{\mathsf{S\,n}}(n-k))$.

Using these facts, we can now prove the main result of this section.

Theorem 5.24. *For all $n : \mathbb{N}$, $\mathsf{Lehmer}(n) \simeq \mathsf{Aut}(\mathsf{Fin}_{\mathsf{S\,n}})$.*

Proof. For $n = 0$, note that $\mathsf{Lehmer}(0)$ is contractible, and so is $\mathsf{Aut}(\mathsf{Fin}_{\mathsf{S\,0}})$, hence we have that $\mathsf{Lehmer}(0) \simeq \mathbf{1} \simeq \mathsf{Aut}(\mathsf{Fin}_{\mathsf{S\,0}})$. For $n = \mathsf{S}\,m$, we compute the following chain of equivalences.

$$\begin{array}{llr}
 & \mathsf{Lehmer}(\mathsf{S}\,m) & \\
\simeq & \mathsf{Fin}_{\mathsf{S\,S\,m}} \times \mathsf{Lehmer}(m) & \text{by definition} \\
\simeq & \mathsf{Fin}_{\mathsf{S\,S\,m}} \times \mathsf{Aut}(\mathsf{Fin}_{\mathsf{S\,m}}) & \text{induction hypothesis} \\
\simeq & \sum_{k:\mathsf{Fin}_{\mathsf{S\,S\,m}}} \mathsf{Fin}_{\mathsf{S\,m}} \simeq \mathsf{Fin}_{\mathsf{S\,m}} & \Sigma \text{ over a constant family} \\
\simeq & \sum_{k:\mathsf{Fin}_{\mathsf{S\,S\,m}}} \mathsf{Fin}^{-}_{\mathsf{S\,S\,m}}(m) \simeq \mathsf{Fin}_{\mathsf{S\,m}} & \text{by Lemma 5.23 part 1} \\
\simeq & \sum_{k:\mathsf{Fin}_{\mathsf{S\,S\,m}}} \mathsf{Fin}^{-}_{\mathsf{S\,S\,m}}(m) \simeq \mathsf{Fin}^{-}_{\mathsf{S\,S\,m}}(m-k) & \text{by Lemma 5.23 part 1} \\
\simeq & \mathsf{Aut}(\mathsf{Fin}_{\mathsf{S\,S\,m}}) & \text{by Lemma 5.23 part 2}
\end{array}$$

□

By composing Theorem 5.24 and Corollary 5.21, we obtain the final equivalence.

Corollary 5.25. *For all $n : \mathbb{N}$, $\mathsf{S}_{\mathsf{n}} \simeq \mathsf{Lehmer}(n) \simeq \mathsf{Aut}(\mathsf{Fin}_{\mathsf{S\,n}})$.*

6 CORRESPONDENCE BETWEEN Π AND $\mathcal{U}_{\mathsf{Fin}}$

In this section, we first translate Π to its additive fragment Π^{+}. This is the language that we interpret to $\mathcal{U}_{\mathsf{Fin}}$ and back, using the tools developed in the previous sections. Further, we go through an intermediate step of the language $\Pi^{\wedge}$, which is a simplified variant of Π^{+} that uses adjacent transpositions for combinators, while preserving all the required structure.

$$\begin{tikzcd}
\Pi \arrow[r, "\mathsf{eval}"] & \Pi^{+} \arrow[r, bend left, "\mathsf{eval}^{+}"] & \Pi^{\wedge} \arrow[l, bend left, "\mathsf{quote}^{+}"] \arrow[r, bend left, "\mathsf{eval}^{\wedge}"] & \mathcal{U}_{\mathsf{Fin}} \arrow[l, bend left, "\mathsf{quote}^{\wedge}"]
\end{tikzcd}$$

We present the types and 1-combinators of Π^{+} and $\Pi^{\wedge}$ in Figures 8 and 9 respectively, eliding the 2-combinators for brevity. We enforce that there is a unique 2-combinator between compatible 1-combinators, by relating them with a truncation. These can be found in Appendix A.1 and in the accompanying Agda code.

The translations between the languages are defined separately on types, 1-combinators, and 2-combinators. Following the terminology of Normalisation by Evaluation (NbE), the translations

$$\begin{array}{rrcll}
\mathsf{id}{\leftrightarrow}_1 : & A & \leftrightarrow^+ & A & : \mathsf{id}{\leftrightarrow}_1 \\
\mathsf{unite}_+\mathsf{l} : & \mathbb{0} + A & \leftrightarrow^+ & A & : \mathsf{uniti}_+\mathsf{l} \\
\mathsf{swap}_+ : & A + B & \leftrightarrow^+ & B + A & : \mathsf{swap}_+ \\
\mathsf{assocl}_+ : & A + (B + C) & \leftrightarrow^+ & (A + B) + C & : \mathsf{assocr}_+
\end{array}$$

$$\frac{\vdash c_1 : A \leftrightarrow^+ B \quad \vdash c_2 : B \leftrightarrow^+ C}{\vdash c_1 \circledcirc c_2 : A \leftrightarrow^+ C} \qquad \frac{\vdash c_1 : A \leftrightarrow^+ B \quad \vdash c_2 : C \leftrightarrow^+ D}{\vdash c_1 \oplus c_2 : A + C \leftrightarrow^+ B + D}$$

Fig. 8. Π^+ syntax

from the left to the right, going from the syntax towards the semantics, are called eval and the translations the other way are called quote.

To state our results formally, we organise the syntax for each language using its syntactic category. We define them formally in the appendix, and only state our results here. For each of the Π, Π^+ and $\Pi^\wedge$ languages, their syntactic categories, respectively $\mathbf{\Pi}_{\text{cat}}$, $\mathbf{\Pi}^+_{\text{cat}}$ and $\mathbf{\Pi}^\wedge_{\text{cat}}$, have 0-cells for types (U, U^+, and $\mathbb{N}$), 1-cells for 1-combinators ($\leftrightarrow$, $\leftrightarrow^+$, and $\leftrightarrow^\wedge$), and 2-cells for 2-combinators ($\Leftrightarrow$, $\Leftrightarrow^+$, and $\Leftrightarrow^\wedge$). These syntactic categories here are actually $(2,0)$-categories, since all the 1-cells and 2-cells are invertible. They are also locally-strict, or locally-posetal, because there is at most one 2-cell between compatible 1-cells.

We use the eval/quote translation maps to construct functors between these categories. We only name the maps on the 0, 1, and 2-cells – the coherences hold by definition or by calculation, which is shown in our accompanying Agda code. We use these functors to state our results establishing the equivalences between the languages.

6.1 Π to Π^+

First, we show how to translate Π programs to Π^+, which is the additive fragment of Π. The syntax for 1-combinators is given in Figure 8.

Π has two 0-ary type constructors, and two binary type constructors – the additive tensor product and the multiplicative one. Π^+ has all the type constructors of Π except multiplication. However, we will show how to recover the multiplicative structure, by defining multiplication as repeated addition. We encode $\times$ in terms of $+$ as follows.

Definition 6.1 ($\times : U^+ \to U^+ \to U^+$).

$$\mathbb{0} \times Y \triangleq \mathbb{0}$$

$$\mathbb{1} \times Y \triangleq Y$$

$$(X_1 + X_2) \times Y \triangleq X_1 \times Y + X_2 \times Y$$

Lemma 6.2. *There are two symmetric monoidal structures on $\mathbf{\Pi}^+_{\text{cat}}$, given by $(\mathbb{0}, +)$ and $(\mathbb{1}, \times)$, with $\times$ distributing over $+$, giving it a rig structure.*

Using this rig structure, we translate Π to Π^+, constructing a rig equivalence from $\mathbf{\Pi}_{\text{cat}}$ to $\mathbf{\Pi}^+_{\text{cat}}$.

Definition 6.3 (eval).

$$\begin{aligned}
&\mathsf{eval}_0 : U \to U^+ \\
&\mathsf{eval}_1 : (c : X \leftrightarrow Y) \to \mathsf{eval}_0(X) \leftrightarrow \mathsf{eval}_0(Y) \\
&\mathsf{eval}_2 : (\alpha : p \Leftrightarrow q) \to \mathsf{eval}_1(p) \Leftrightarrow \mathsf{eval}_1(q)
\end{aligned}$$

Theorem 6.4. eval *gives a rig equivalence between $\mathbf{\Pi}_{\text{cat}}$ and $\mathbf{\Pi}^+_{\text{cat}}$.*

$$\begin{array}{rcccl} \mathrm{id}{\leftrightarrow_1}: & n & \leftrightarrow^{\wedge} & n & : \mathrm{id}{\leftrightarrow_1} \\ \mathrm{swap}: & \mathsf{S}\,\mathsf{S}\,n & \leftrightarrow^{\wedge} & \mathsf{S}\,\mathsf{S}\,n & : \mathrm{swap} \end{array}$$

$$\frac{\vdash c_1 : n \leftrightarrow^{\wedge} m \quad \vdash c_2 : m \leftrightarrow^{\wedge} o}{\vdash c_1 \circledcirc c_2 : n \leftrightarrow^{\wedge} o} \qquad \frac{\vdash c : n \leftrightarrow^{\wedge} m}{\vdash \oplus(c) : \mathsf{S}\,n \leftrightarrow^{\wedge} \mathsf{S}\,m}$$

Fig. 9. $\Pi^{\wedge}$ syntax

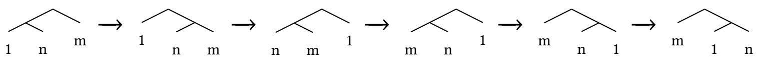


Fig. 10. Braiding from transpositions, recursive case

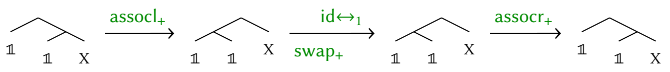


Fig. 11. Transpositions from braiding

6.2 Π^+ to $\Pi^{\wedge}$

Next, we show how to translate Π^+ programs to $\Pi^{\wedge}$ and back. $\Pi^{\wedge}$ is a simplified variant of Π^+, with (unary) natural numbers for 0-cells, 1-combinators generated by adjacent transpositions, and an appropriate set of 2-combinators. We give the syntax, again omitting 2-combinators, in Figure 9.

As described, $\Pi^{\wedge}$ doesn't have a tensor product, but we can build it simply by adding up natural numbers, and, we need to verify that this indeed equips $\mathbf{\Pi}^{\wedge}_{\mathrm{cat}}$ with a symmetric monoidal structure. Since each object is a natural number, this makes $\mathbf{\Pi}^{\wedge}_{\mathrm{cat}}$ a *PROP*, that is, a products and permutations category.

To produce a braiding $n + m \leftrightarrow^{\wedge} m + n$ from adjacent transpositions, we recursively traverse the left subexpression, swapping each element using adjacent transpositions along the elements on the right, placing it in the correct position. The computational content of this translation can be visualised using tree transformations, for the recursive case in Figure 10. The challenging part is showing that these moves are coherent with respect to 2-combinators.

Lemma 6.5. $\mathbf{\Pi}^{\wedge}_{\mathrm{cat}}$ *has a symmetric monoidal structure, with the unit given by 0 and the tensor product given by natural number addition.*

Using this symmetric monoidal structure, we translate from Π^+ to $\Pi^{\wedge}$.

Definition 6.6 (eval^+).

$$\begin{aligned} &\mathrm{eval}^+_0 : U^+ \to \mathbb{N} \\ &\mathrm{eval}^+_1 : (c : t_1 \leftrightarrow^+ t_2) \to \mathrm{eval}^+_0(t_1) \leftrightarrow^{\wedge} \mathrm{eval}^+_0(t_2) \\ &\mathrm{eval}^+_2 : (\alpha : c_1 \Leftrightarrow^+ c_2) \to \mathrm{eval}^+_1(c_1) \Leftrightarrow^{\wedge} \mathrm{eval}^+_1(c_2) \end{aligned}$$

To go back from $\Pi^{\wedge}$ to Π^+, we turn a natural number into a Π^+ type, using right-biased addition, that is, the natural number n gets mapped to the type $\mathbb{1} + (\mathbb{1} + (\mathbb{1} + \ldots + \mathbb{0}))$. Since the types are already right-biased, an adjacent transposition in $\Pi^{\wedge}$ is easily encoded by using the braiding in Π^+, as shown in Figure 11. Again, these are shown to be coherent.

Definition 6.7 (quote^+).

$$\begin{aligned}
&\mathsf{quote}^+_0 : \mathbb{N} \to U^+\\
&\mathsf{quote}^+_1 : (p : X_1 \leftrightarrow^\wedge X_2) \to \mathsf{quote}^+_0(X_1) \leftrightarrow \mathsf{quote}^+_0(X_2)\\
&\mathsf{quote}^+_2 : (\alpha : p_1 \Leftrightarrow^\wedge p_2) \to \mathsf{quote}^+_1(p_1) \Leftrightarrow \mathsf{quote}^+_1(p_2)
\end{aligned}$$

Theorem 6.8. $\mathsf{eval}^+/\mathsf{quote}^+$ *give a symmetric monoidal equivalence between* $\mathbf{\Pi}^+_{\mathsf{cat}}$ *and* $\mathbf{\Pi}^\wedge_{\mathsf{cat}}$.

6.3 $\Pi^\wedge$ to $\mathcal{U}_{\mathsf{Fin}}$

Finally, we show how to interpret $\Pi^\wedge$ to $\mathcal{U}_{\mathsf{Fin}}$, and back from $\mathcal{U}_{\mathsf{Fin}}$ to $\Pi^\wedge$. Types in $\Pi^\wedge$ are interpreted as 0-cells in $\mathcal{U}_{\mathsf{Fin}}$, that is, a natural number n is mapped to Fin_n. The 1-combinators in $\Pi^\wedge$ are mapped to 1-paths in $\mathcal{U}_{\mathsf{Fin}}$, that is, 1-loops in each connected component, equivalent to $\mathsf{Aut}(\mathsf{Fin}_n)$. In $\Pi^\wedge$, the 1-combinators are generated by adjacent transpositions, so these can be mapped to words in S_n and then to automorphisms using Corollary 5.25. Finally, 2-combinators are mapped to 2-paths between loops in $\mathcal{U}_{\mathsf{Fin}}$.

Definition 6.9 ($\mathsf{eval}^\wedge$).

$$\begin{aligned}
&\mathsf{eval}^\wedge_0 : \mathbb{N} \to \mathcal{U}_{\mathsf{Fin}}\\
&\mathsf{eval}^\wedge_1 : (c : t_1 \leftrightarrow^\wedge t_2) \to \mathsf{eval}^\wedge_0(t_1) = \mathsf{eval}^\wedge_0(t_2)\\
&\mathsf{eval}^\wedge_2 : (\alpha : c_1 \Leftrightarrow^\wedge c_2) \to \mathsf{eval}^\wedge_1(c_1) = \mathsf{eval}^\wedge_1(c_2)
\end{aligned}$$

0-cells in $\mathcal{U}_{\mathsf{Fin}}$ are mapped to their cardinalities in $\Pi^\wedge$, 1-loops are decoded to words in S_n to generate a sequence of adjacent transpositions, producing a 1-combinator in $\Pi^\wedge$. Finally, 2-paths are quoted back to 2-combinators in $\Pi^\wedge$.

Definition 6.10 ($\mathsf{quote}^\wedge$).

$$\begin{aligned}
&\mathsf{quote}^\wedge_0 : \mathcal{U}_{\mathsf{Fin}} \to \mathbb{N}\\
&\mathsf{quote}^\wedge_1 : (p : X_1 = X_2) \to \mathsf{quote}^\wedge_0(X_1) \leftrightarrow^\wedge \mathsf{quote}^\wedge_0(X_2)\\
&\mathsf{quote}^\wedge_2 : (\alpha : p_1 = p_2) \to \mathsf{quote}^\wedge_1(p_1) \Leftrightarrow^\wedge \mathsf{quote}_1(p_2)
\end{aligned}$$

Theorem 6.11. $\mathsf{eval}^\wedge/\mathsf{quote}^\wedge$ *give a symmetric monoidal equivalence between* $\mathbf{\Pi}^\wedge_{\mathsf{cat}}$ *and* $\mathcal{U}_{\mathsf{Fin}}$.

The semantics that we presented here takes a different route to constructing the permutation from a Π combinator, compared to the direct interpretation given using the big-step interpreter in Section 3.2. We verify that the two semantics agree, establishing that the semantics is adequate and fully abstract.

Definition 6.12 ($(\!| - |\!)$).

$$\begin{aligned}
&(\!| - |\!)_0 : U \to \mathcal{U}_{\mathsf{Fin}} && (\!| - |\!)_1 : (c : X \leftrightarrow Y) \to (\!| X |\!)_0 =_{\mathcal{U}_{\mathsf{Fin}}} (\!| Y |\!)_0\\
&(\!| - |\!)_0 \triangleq \mathsf{eval}^\wedge_0 \circ \mathsf{eval}^+_0 \circ \mathsf{eval}_0 && (\!| - |\!)_1 \triangleq \mathsf{eval}^\wedge_1 \circ \mathsf{eval}^+_1 \circ \mathsf{eval}_1
\end{aligned}$$

Theorem 6.13 (Full Abstraction and Adequacy). *For any* $c_1, c_2 : X \leftrightarrow Y$, *we have that*

$$[\![c_1]\!] = [\![c_2]\!] \textit{ if and only if } (\!| c_1 |\!)_1 = (\!| c_2 |\!)_1$$

7 APPLICATIONS TO REVERSIBLE CIRCUITS

Using our semantics, we can normalise, synthesise, prove equivalence, and reason about Π programs. We describe a few applications in this section, and a more comprehensive list of applications and examples along with their formalisation is available in our artifact [Choudhury et al. 2021a].

7.1 Normalisation

Normalisation happens by evaluating a combinator to a permutation and then reifying it back. The evaluation step goes from Π to Π^+ reducing $\times$ to $+$ and translating the corresponding combinators, then from Π^+ to $\Pi^\wedge$ reducing them to natural numbers and adjacent transpositions, then finally to $\mathcal{U}_{\mathsf{Fin}}$ through words in S_n and Lehmer codes. Finally, the quotation step goes all the way back until Π^+. We define this composite norm function below using our previous evaluation function $(\!| - |\!)$.

Definition 7.1 (Normalisation of Π programs).

$$\mathsf{norm}_0 : U \longrightarrow U^+ \qquad \mathsf{norm}_1 : (c : X \leftrightarrow Y) \longrightarrow \mathsf{norm}_0(X) \leftrightarrow^+ \mathsf{norm}_0(Y)$$

$$\mathsf{norm}_0 = \mathsf{quote}^+_0 \circ \mathsf{quote}^\wedge_0 \circ (\!| - |\!)_0 \qquad \mathsf{norm}_1 = \mathsf{quote}^+_1 \circ \mathsf{quote}^\wedge_1 \circ (\!| - |\!)_1$$

As an example, consider the reversibleOr1 circuit introduced in Section 3. After applying norm_1, the type $\mathbb{B}\ 3$ normalises to $\mathbb{8}+ \equiv \mathbb{1} + (\mathbb{1} + (\mathbb{1} + (\mathbb{1} + (\mathbb{1} + (\mathbb{1} + (\mathbb{1} + (\mathbb{1} + \mathbb{0})))))))$, and the circuit reduces to the following normal form:

```
reversibleOrNorm : 𝟠+ ↔₁₊ 𝟠+
reversibleOrNorm = (id↔₁ ⊕ id↔₁ ⊕ assocl₊ ⊚ (swap₊ ⊕ id↔₁) ⊚ assocr₊) ⊚
                   (id↔₁ ⊕ assocl₊ ⊚ (swap₊ ⊕ id↔₁) ⊚ assocr₊) ⊚
                   (assocl₊ ⊚ (swap₊ ⊕ id↔₁) ⊚ assocr₊) ⊚
                   - elided
```

7.2 Equivalence

By normalisation, we also get a decision procedure for program equivalence – two circuits are equivalent iff they represent the same permutation, or equivalently, they have the same normal form. Recall that in Section 2 we introduced two different ways of computing reversible disjunction using two Qiskit circuit implementations, and then in Section 3 showed the corresponding Π definitions: reversibleOr1 and reversibleOr2.

After applying norm_1 to each circuit, we can verify that they both reduce to the same normal form reversibleOrNorm, and there exist 2-combinators between each of them and the normal form. More general, user-guided, reasoning can be done using the sound and complete level-2 combinators to rewrite Π programs.

7.3 Synthesis and Verification

Instead of manually writing Π programs to implement the reversible disjunction specification, it is possible to simply write down the permutation directly and then quote it, synthesising a Π program. We show this below, using a function lookup to write tabulated permutations for readability.

```
reversibleOrPerm : Aut (Fin 8)
reversibleOrPerm = equiv f f f-f f-f
  where f : Fin 8 → Fin 8
        f = lookup (0 :: 5 :: 6 :: 7 :: 4 :: 1 :: 2 :: 3 :: nil)
        f-f : (x : Fin 8) → f (f x) == x
        - elided
```

The permutation uses the canonical encoding of sequences of bits as natural numbers, e.g., (false, true, true) is encoded as 011 or 3. The second entry in the lookup table maps index 1 (= 001) to the value 5 (= 101), following the reversible disjunction specification (recall the definition $\mathsf{reversibleOr}(h, b_1, b_2) = (h \underline{\vee} (b_1 \vee b_2),\ b_1,\ b_2)$ where $\vee$ is disjunction and $\underline{\vee}$ is exclusive-disjunction). Quoting this permutation generates the same normalised program reversibleOrNorm, matching the desired structured type of a 1-combinator on $\mathbb{8}+$.

Similarly, we can also verify whether a circuit is correctly implemented – by running the equivalence described above in the other direction, we can evaluate the circuit to a bijection and compare it extensionally with the intended one.

7.4 Reasoning and Transfer of Theorems

In our development, we described a number of different representations of permutations: Π, Π^+ and $\Pi^\wedge$ programs, lists of transpositions, Lehmer codes, and automorphisms of finite sets. The equivalences between these representations allow us to transfer theorems about permutations – we prove them on the most suitable representation, and then transport them to a different representation "for free".

As an example, consider the following problem. The Toffoli gate on 3 bits can be extended to a reversible gate on 4 bits, by inserting a wire in four possible positions. This gate performs identity on the newly inserted bit, and controlled-controlled not on the three remaining ones. We show how to write these gates using cif.

```
toffoli₃¹ toffoli₃² toffoli₃³ toffoli₃⁴ : 𝔹 4 ↔₁ 𝔹 4
toffoli₃¹ = cif (cif (swap₊ ⊗ id↔₁) (id↔₁ ⊗ id↔₁)) (id↔₁ ⊗ (id↔₁ ⊗ id↔₁))
toffoli₃² = cif (cif (id↔₁ ⊗ swap₊) (id↔₁ ⊗ id↔₁)) (id↔₁ ⊗ (id↔₁ ⊗ id↔₁))
toffoli₃³ = cif (cif (id↔₁ ⊗ id↔₁) (swap₊ ⊗ id↔₁)) (id↔₁ ⊗ (id↔₁ ⊗ id↔₁))
toffoli₃⁴ = cif (cif (id↔₁ ⊗ id↔₁) (id↔₁ ⊗ swap₊)) (id↔₁ ⊗ (id↔₁ ⊗ id↔₁))
```

On the other hand, it is possible to construct a proper 4-bit toffoli gate.

```
toffoli₄ : 𝔹 4 ↔₁ 𝔹 4
toffoli₄ = controlled (controlled (controlled not))
```

Now, we can ask: is it possible to implement toffoli₄ gate using only toffoli₃ gates?

Solving this problem in the circuit representation is difficult. However, we can observe that all toffoli₃ gates implement even permutations, that is, permutations that contain an even number of transpositions. On the other hand, toffoli₄ implements an odd permutation. It can be easily proved that any composition of even permutations always produces an even permutation, which answers the question to the negative. Thus, we formulate the theorem using the representation of lists of transpositions, and transfer it to circuits using the equivalences proven in the previous sections.

We define a function $\mathsf{parity} : (X \leftrightarrow Y) \longrightarrow \mathbf{2}$ to compute the parity of a permutation defined by a combinator c, by first transforming a Π program into a list of transpositions (using $\mathsf{eval}_1^\wedge$ from Definition 6.9), and then computing the parity of this representation, which amounts to a simple checking of the parity of the length of the list.

Proposition 7.2. *For two Π circuits c_1, c_2, if $c_1 \Leftrightarrow c_2$, then* $\mathsf{parity}(c_1) = \mathsf{parity}(c_2)$.

Using Proposition 7.2, we complete the proof by computing the parity of toffoli₄, and all combinations of the toffoli₃ gates.

8 DISCUSSION & RELATED WORK

The main theme of our work is the semantic foundation of reversible languages. We prove that a programming language presentation of reversible computing based on algebraic types matches *exactly* the categorified group-theoretic semantics, thereby closing the circle on a complete Curry-Howard-Lambek correspondence for reversible languages. Historically, the first such correspondence related the λ-calculus, intuitionistic logic, and cartesian-closed categories [Curry et al. 1980]. For reversible languages, the Curry-Howard correspondence was suggested by Sparks and Sabry [2014], and the Lambek correspondence was suggested by Carette and Sabry [2016] and Carette, James, and Sabry

Table 1. Summary of the established correspondences

Π	$\bigsqcup_n \mathcal{B}\mathsf{S_n}$	$\mathcal{B}$	$\mathcal{U}_{\mathsf{Fin}}$
Types	Natural numbers	Finite sets	0-cells
1-combinators	Generators of $\mathsf{S_n}$	Bijections	1-paths
2-combinators	Relations of $\mathsf{S_n}$	Homotopies	2-paths

[2021]. For a 1-bit fragment of Π, this was established by Carette, Chen, Choudhury, and Sabry [2018]. In this work, we have completely established this correspondence, as summarised in Table 1. In the remainder of this section, we discuss some broader related work.

Coherence and Rewriting. Higher-order term-rewriting systems and word problems have a long history of being formalised in proof assistants like homotopy.io, Agda, Coq and Lean. As part of the proof of our main result, we developed a rewriting system for the Coxeter relations for $\mathsf{S_n}$ to solve its word problem. Hiver [2021] describes an explicit algorithm for producing normal forms, which could provide an alternative to our rewriting system. Other encodings of permutations as listed vectors, matrices, inductively generated trees (Motzkin trees), Young diagrams, or string diagrams, proved either difficult to formalise in type theory or difficult to relate directly to the primitive type isomorphisms of Π. The automatic Knuth and Bendix [1970] completion produced too many equations making proving correctness and termination intractable.

Coherence theorems are famous problems in category theory, and Mac Lane's coherence theorem [Gurski and Osorno 2013; Joyal and Street 1993; MacLane 1963] for monoidal categories is a particular one. The use of rewriting and proof assistants to prove coherence theorems for higher categorical structures has a long history, see [Beylin and Dybjer 1996; Forest and Mimram 2018].

Computing with Univalence. In HoTT, univalence characterises the path type in the universe as equivalences of types. The map $\mathsf{idtoeqv} : A =_{\mathcal{U}} B \longrightarrow A \simeq B$ is constructed using path induction. The term $\mathsf{ua} : A \simeq B \longrightarrow A =_{\mathcal{U}} B$, its computation rule $\mathsf{ua}_\beta : (e : A \simeq B) \longrightarrow \mathsf{idtoeqv}(\mathsf{ua}(e)) = e$, and its extensionality rule $\mathsf{ua}_\eta : (p : A =_{\mathcal{U}} B) \longrightarrow p = \mathsf{ua}(\mathsf{idtoeqv}(\mathrm{p}))$ are generally added as postulates when formalising in Agda. Together, ua and ua_β give the full univalence axiom $(A \simeq B) \simeq (A =_{\mathcal{U}} B)$ [Orton and Pitts 2018, Theorem 3.5]. By giving a computable presentation for a univalent subuniverse, we are able to describe its path space syntactically via a complete equational axiomatisation of the equivalences between types in the subuniverse. Univalent typoids [Petrakis 2019] are another way of axiomatising univalent subuniverses.

In the subuniverse of finite types, $\mathsf{idtoeqv}$ corresponds to giving a denotation for a program (1-combinator), which is easily done by induction. The ua map corresponds to synthesising a program from an equivalence (which, in general, is of course undecidable [Krogmeier et al. 2020]). In case of reversible boolean circuits, it is decidable, as we have shown, but still far from trivial, which matches the need to assert the existence of ua without giving a constructive argument. Then, the computation rule ua_β expresses the fact that program synthesis is sound, while ua_η corresponds to the soundness of the equational theory (Π 2-combinators). Thus, our results suggest a new way of thinking about computing using the univalence principle, which provides an intuition on why certain constructions are hard (or impossible in the general case). There are other, different approaches to computing with univalence, in [Angiuli et al. 2021; Tabareau et al. 2021], and in Cubical Type Theory[Angiuli 2019; Vezzosi et al. 2019].

Algebraic Theories. In universal algebra, algebraic theories are used to describe structures such as groups or rings. A specific group or ring is a model of the appropriate algebraic theory. Algebraic theories are usually *presented* in terms of logical syntax, that is, as first-order theories whose signatures allow only functional symbols, and whose axioms are universally quantified equations. In his seminal thesis, Lawvere [1963] defined a presentation-free categorical notion of universal algebraic structure, called a Lawvere theory. Programming Languages, such as the λ-calculus, can be viewed as algebraic structures with variable-binding operators, which can be formalised using second-order algebraic theories [Fiore and Mahmoud 2010], or algebraic theories with closed structure [Hyland 2017], called λ-theories, making the λ-calculus the presentation of the initial λ-theory Λ. Our family of reversible languages Π have been presented as first-order algebraic 2-theories [Beke 2011; Cohen 2009; Yanofsky 2000], which are a categorification of algebraic theories. The types $\mathbb{0}$ and $\mathbb{1}$ are nullary function symbols, the type formers $+$ and $\times$ are binary function symbols, the 1-combinators are invertible reduction rules, and the 2-combinators are equations or coherence diagrams of compositions of reduction rules. Just like models of Lawvere theories are given by algebras of (finitary) monads on $\mathcal{S}$et, models of 2-theories are given by algebras of 2-monads on $\mathcal{C}$at. Our development is related to the free symmetric monoidal completion 2-monad.

Free Symmetric Monoidal Category. The free symmetric monoidal category has been used to study concurrency [Hyland and Power 2004], Petri nets [Baez et al. 2021], combinatorial structures [Fiore et al. 2008], quantum mechanics [Abramsky 2005a], and bicategorical models of (differential) linear logic [Melliès 2019]. The forgetful functor from $\mathcal{S}$ymMonCat, the 2-category of (small) symmetric monoidal categories, strong symmetric monoidal functors, and monoidal natural transformations, to the 2-category $\mathcal{C}$at, has a left adjoint giving the free symmetric monoidal category $\mathcal{F}_{\text{SM}}(C)$ on a category C. This is a 2-monad on $\mathcal{C}$at [Blackwell et al. 1989], whose algebras are (strict) symmetric monoidal categories. A construction of $\mathcal{F}_{\text{SM}}$ is given by Abramsky [2005a]. Concretely, the objects of $\mathcal{F}_{\text{SM}}(C)$ are given by lists of objects of C, that is, a pair $(n : \mathbb{N}, A : [n] \longrightarrow C_0)$; morphisms between (n, A) and (n, B) are pairs (π, λ) where π is a permutation of $[n]$, and $\lambda_i : A_i \longrightarrow B_{\pi(i)}$ for $1 \leq i \leq n$. Abstractly, this is given by the Grothendieck construction $\int F$ of the functor $F : \mathcal{B} \longrightarrow \mathcal{C}$at from the groupoid of finite sets and bijections to $\mathcal{C}$at, assigning each natural number n to the n-power C^n of C, and each permutation on $[n]$ inducing an endofunctor on C^n by action. The groupoid $\mathcal{B}$ is the free symmetric monoidal category (groupoid) on one generator, $\mathcal{F}_{\text{SM}}(\mathbf{1})$.

Coherence and normalisation problems for monoids in constructive type theory using coherence for monoidal categories were studied by Beylin and Dybjer [1996]. In HoTT, coherence for the free monoidal groupoid over a groupoid and the proof of its universal property has been considered by Piceghello [2020]. Free commutative monoids in type theory have been studied by Gylterud [2020], and in HoTT by Choudhury and Fiore [2019, 2021]. The free symmetric monoidal groupoid $\mathcal{F}_{\text{SM}}(A)$ over a groupoid A can be given by $\sum_{X:\mathcal{U}_{\text{Fin}}} A^X$, or it can be presented as an algebraic 2-theory using 1-HITs. These HITs and the proof of their universal property have been considered by Choudhury and Fiore [2019]; Piceghello [2019]. The proof of the universal property of $\mathcal{F}_{\text{SM}}$ is asserted by appealing to Mac Lane's coherence theorem for symmetric monoidal categories, and using the fact that the finite symmetric group S_n encodes the group of permutations on a finite set. The existence of the proof is folklore. We have produced a proof and formalised it in constructive type theory. A stronger universal property that $\mathcal{B}$ is biinitial in RigCat, is in [Elgueta 2021].

Reversible Programming Languages. The practice of programming languages is replete with *ad hoc* instances of reversible computations: database transactions, mechanisms for data provenance, checkpoints, stack and exception traces, logs, backups, rollback recoveries, version control systems, reverse engineering, software transactional memories, continuations, backtracking search, and multiple-level undo features in commercial applications. In the early nineties, Baker [1992a,b] argued for a systematic, first-class, treatment of reversibility. But intensive research in full-fledged reversible models of computations and reversible programming languages was only sparked by the discovery of deep connections between physics and computation [Bennett and Landauer 1985; Frank 1999; Fredkin and Toffoli 1982; Hey 1999; Landauer 1961; Peres 1985; Toffoli 1980], and by the potential for efficient quantum computation [Feynman 1982]. The early developments of reversible programming languages started with a conventional programming language, e.g., an extended λ-calculus, and either (i) extended the language with a history mechanism [Danos and Krivine 2004; Huelsbergen 1996; Kluge 2000; van Tonder 2004], or (ii) imposed constraints on the control flow constructs to make them reversible [Yokoyama and Glück 2007]. More modern approaches recognize that reversible programming languages require a fresh approach and should be designed from first principles without the detour via conventional irreversible languages [Abramsky 2005b; Di Pierro et al. 2006; Mu et al. 2004; Yokoyama et al. 2008]. The version of Π studied in this paper is restricted to finite types and terminating total computations. It would be interesting to understand which versions of free monoidal structures correspond to extensions of Π with recursive types [Bowman et al. 2011; James and Sabry 2012], or negative and fractional types [Chen, Choudhury, Carette, and Sabry 2020; Chen and Sabry 2021].

Permutations. Finding formal systems for expressing various flavors of computable functions has been a major focus of logic and computer science since its inception. Permutations, being at the core of reversible computing, are an interesting class of functions, for which there are few formal systems. The system we have developed brings in all the associated benefits of syntactic calculi, notably, their calculational flavor. Instead of comparing two reversible programs by extensional equality of the underlying bijective functions, a calculus offers more nuanced techniques that can enforce additional intensional constraints on the desired equality relation.

DATA AVAILABILITY STATEMENT

The accompanying Agda formalisation for the paper including the full syntax and proofs is available at the following GitHub repository: https://github.com/vikraman/popl22-symmetries-artifact [Choudhury, Karwowski, and Sabry 2021a].

ACKNOWLEDGMENTS

The authors would like to thank Jacques Carette and Chao-Hong Chen for numerous discussions, ideas, and contributions. Robert Rose and Siva Somayyajula contributed to a previous version of this work. The first author is grateful to Marcelo Fiore for various discussions about the free symmetric monoidal groupoid, and Thorsten Alteknirch, Andrew Pitts, Jamie Vicary, Eric Finster, Nathanael Arkor, and Younesse Kaddar, for related discussions. The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions. This paper is dedicated to the memory of Socrates, the demure cat.

REFERENCES

Scott Aaronson, Daniel Grier, and Luke Schaeffer. 2017. The Classification of Reversible Bit Operations. In *8th Innovations in Theoretical Computer Science Conference (ITCS 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 67)*, Christos H. Papadimitriou (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 23:1–23:34. https://doi.org/10.4230/LIPIcs.ITCS.2017.23

Samson Abramsky. 2005a. Abstract Scalars, Loops, and Free Traced and Strongly Compact Closed Categories. In *Algebra and Coalgebra in Computer Science (Lecture Notes in Computer Science)*, José Luiz Fiadeiro, Neil Harman, Markus Roggenbach, and Jan Rutten (Eds.). Springer, Berlin, Heidelberg, 1–29. https://doi.org/10.1007/11548133_1

Samson Abramsky. 2005b. A Structural Approach to Reversible Computation. *Theoretical Computer Science* 347, 3 (Dec. 2005), 441–464. https://doi.org/10.1016/j.tcs.2005.07.002

Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose Martín-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O'Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyanov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. 2019. Qiskit: An Open-Source Framework for Quantum Computing. Zenodo. https://doi.org/10.5281/ZENODO.2562110

Carlo Angiuli. 2019. *Computational Semantics of CartesianCubical Type Theory*. Ph. D. Dissertation. Carnegie Mellon University, Pittsburgh, PA. https://www.cs.cmu.edu/~cangiuli/thesis/thesis.pdf

Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. 2021. Internalizing Representation Independence with Univalence. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 12:1–12:30. https://doi.org/10.1145/3434293

John C. Baez and James Dolan. 2001. From Finite Sets to Feynman Diagrams. In *Mathematics Unlimited — 2001 and Beyond*, Björn Engquist and Wilfried Schmid (Eds.). Springer, Berlin, Heidelberg, 29–50. https://doi.org/10.1007/978-3-642-56478-9_3

John C. Baez, Fabrizio Genovese, Jade Master, and Michael Shulman. 2021. Categories of Nets. *arXiv:2101.04238 [cs, math]* (April 2021). arXiv:2101.04238 [cs, math] http://arxiv.org/abs/2101.04238

John C Baez, Alexander E Hoffnung, and Christopher D Walker. 2010. HIGHER DIMENSIONAL ALGEBRA VII: GROUPOIDIFICATION. *Theory and Applications of Categories* 24 (2010), 66. http://www.tac.mta.ca/tac/volumes/24/18/24-18abs.html

Henry G. Baker. 1992a. Lively Linear Lisp: "Look Ma, No Garbage!". *ACM SIGPLAN Notices* 27, 8 (Aug. 1992), 89–98. https://doi.org/10.1145/142137.142162

Henry G. Baker. 1992b. NREVERSAL of Fortune — The Thermodynamics of Garbage Collection. In *Memory Management (Lecture Notes in Computer Science)*, Yves Bekkers and Jacques Cohen (Eds.). Springer, Berlin, Heidelberg, 507–524. https://doi.org/10.1007/BFb0017210

Stephane Beauregard. 2003. Circuit for Shor's Algorithm Using 2n+3 Qubits. *Quantum Information & Computation* 3, 2 (March 2003), 175–185.

Tibor Beke. 2011. Categorification, Term Rewriting and the Knuth–Bendix Procedure. *Journal of Pure and Applied Algebra* 215, 5 (May 2011), 728–740. https://doi.org/10.1016/j.jpaa.2010.06.019

Richard Bellman and Marshall Hall. 1960. *Combinatorial Analysis*. Proceedings of Symposia in Applied Mathematics, Vol. 10. American Mathematical Society. https://doi.org/10.1090/psapm/010

Charles H. Bennett and Rolf Landauer. 1985. The Fundamental Physical Limits of Computation. *Scientific American* 253, 1 (1985), 48–57. https://www.jstor.org/stable/24967723

Sebastian Berger, Andrii Kravtsiv, Gerhard Schneider, and Denis Jordan. 2019. Teaching Ordinal Patterns to a Computer: Efficient Encoding Algorithms Based on the Lehmer Code. *Entropy* 21, 10 (Oct. 2019), 1023. https://doi.org/10.3390/e21101023

Ilya Beylin and Peter Dybjer. 1996. Extracting a Proof of Coherence for Monoidal Categories from a Proof of Normalization for Monoids. In *Types for Proofs and Programs*, Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Stefano Berardi, and Mario Coppo (Eds.). Vol. 1158. Springer Berlin Heidelberg, Berlin, Heidelberg, 47–61. https://doi.org/10.1007/3-540-61780-9_61

Marc Bezem, Ulrik Buchholtz, Pierre Cagne, Bjørn Ian Dundas, and Daniel R. Grayson. 2021. *Symmetry*. https://github.com/UniMath/SymmetryBook

R. Blackwell, G. M. Kelly, and A. J. Power. 1989. Two-Dimensional Monad Theory. *Journal of Pure and Applied Algebra* 59, 1 (July 1989), 1–41. https://doi.org/10.1016/0022-4049(89)90160-6

William J. Bowman, Roshan P. James, and Amr Sabry. 2011. Dagger Traced Symmetric Monoidal Categories and Reversible Programming. In *Workshop on Reversible Computation*.

Ulrik Buchholtz and Egbert Rijke. 2017. The Real Projective Spaces in Homotopy Type Theory. In *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '17)*. IEEE Press, Reykjavík, Iceland, 1–8.

Ulrik Buchholtz, Floris van Doorn, and Egbert Rijke. 2018. Higher Groups in Homotopy Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, Oxford United Kingdom, 205–214. https://doi.org/10.1145/3209108.3209150

Jacques Carette, Chao-Hong Chen, Vikraman Choudhury, and Amr Sabry. 2018. From Reversible Programs to Univalent Universes and Back. *Electronic Notes in Theoretical Computer Science* 336 (April 2018), 5–25. https://doi.org/10.1016/j.

entcs.2018.03.013

Jacques Carette, Roshan P. James, and Amr Sabry. 2021. Embracing the Laws of Physics: Three Reversible Models of Computation. *Advances in Computers* 126 (2021). http://arxiv.org/abs/1811.03678

Jacques Carette and Amr Sabry. 2016. Computing with Semirings and Weak Rig Groupoids. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Vol. 9632. Springer Berlin Heidelberg, Berlin, Heidelberg, 123–148. https://doi.org/10.1007/978-3-662-49498-1_6

Chao-Hong Chen, Vikraman Choudhury, Jacques Carette, and Amr Sabry. 2020. Fractional Types. In *Reversible Computation*. Springer, Cham, 169–186. https://doi.org/10.1007/978-3-030-52482-1_10

Chao-Hong Chen and Amr Sabry. 2021. A Computational Interpretation of Compact Closed Categories: Reversible Programming with Negative and Fractional Types. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 9:1–9:29. https://doi.org/10.1145/3434290

Vikraman Choudhury and Marcelo Fiore. 2019. The Finite-Multiset Construction in HoTT. (Aug. 2019), 40. https://hott.github.io/HoTT-2019/conf-slides/Choudhury.pdf

Vikraman Choudhury and Marcelo Fiore. 2021. Free Commutative Monoids in Homotopy Type Theory. (Oct. 2021). https://arxiv.org/abs/2110.05412v1

Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. 2021a. Artifact for Symmetries in Reversible Programming. Zenodo. https://doi.org/10.5281/zenodo.5671746

Vikraman Choudhury, Jacek Karwowski, and Amr Sabry. 2021b. Symmetries in Reversible Programming: From Symmetric Rig Groupoids to Reversible Programming Languages. (Oct. 2021). https://arxiv.org/abs/2110.05404v1

Dan Christensen. 2015. A Characterization of Univalent Fibrations. (June 2015), 53. http://sweet.ua.pt/dirk/ct2015/slides/Christensen.pdf

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. (2018), 34 pages. https://doi.org/10.4230/LIPICS.TYPES.2015.5

Jonathan Asher Cohen. 2009. Coherence for Rewriting 2-Theories. *arXiv:0904.0125 [cs, math]* (April 2009). arXiv:0904.0125 [cs, math] http://arxiv.org/abs/0904.0125

Haskell B. Curry, J. Roger Hindley, and J. P. Seldin (Eds.). 1980. *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, London ; New York.

Vincent Danos and Jean Krivine. 2004. Reversible Communicating Systems. In *CONCUR 2004 - Concurrency Theory (Lecture Notes in Computer Science)*, Philippa Gardner and Nobuko Yoshida (Eds.). Springer, Berlin, Heidelberg, 292–307. https://doi.org/10.1007/978-3-540-28644-8_19

Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. 2006. Reversible Combinatory Logic. *Mathematical Structures in Computer Science* 16, 4 (Aug. 2006), 621–637. https://doi.org/10.1017/S0960129506005391

Catherine Dubois and Alain Giorgetti. 2018. Tests and Proofs for Custom Data Generators. *Formal Aspects of Computing* 30, 6 (Nov. 2018), 659–684. https://doi.org/10.1007/s00165-018-0459-1

Josep Elgueta. 2021. The Groupoid of Finite Sets Is Biinitial in the 2-Category of Rig Categories. *Journal of Pure and Applied Algebra* 225, 11 (Nov. 2021), 106738. https://doi.org/10.1016/j.jpaa.2021.106738

Richard P. Feynman. 1982. Simulating Physics with Computers. *International Journal of Theoretical Physics* 21, 6 (June 1982), 467–488. https://doi.org/10.1007/BF02650179

M. Fiore, N. Gambino, M. Hyland, and G. Winskel. 2008. The Cartesian Closed Bicategory of Generalised Species of Structures. *Journal of the London Mathematical Society* 77, 1 (Feb. 2008), 203–220. https://doi.org/10.1112/jlms/jdm096

Marcelo Fiore and Ola Mahmoud. 2010. Second-Order Algebraic Theories. In *Mathematical Foundations of Computer Science 2010 (Lecture Notes in Computer Science)*, Petr Hliněný and Antonín Kučera (Eds.). Springer, Berlin, Heidelberg, 368–380. https://doi.org/10.1007/978-3-642-15155-2_33

Marcelo P. Fiore, Roberto Di Cosmo, and Vincent Balat. 2002. Remarks on Isomorphisms in Typed Lambda Calculi with Empty and Sum Types. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS '02)*. IEEE Computer Society, USA, 147.

Simon Forest and Samuel Mimram. 2018. Coherence of Gray Categories via Rewriting. (2018), 16 pages. https://doi.org/10.4230/LIPICS.FSCD.2018.15

Michael P. Frank. 1999. *Reversibility for Efficient Computing.* Ph. D. Dissertation. Massachusetts Institute of Technology, USA.

Edward Fredkin and Tommaso Toffoli. 1982. Conservative Logic. *International Journal of Theoretical Physics* 21, 3 (April 1982), 219–253. https://doi.org/10.1007/BF01857727

Nick Gurski and Angélica M. Osorno. 2013. Infinite Loop Spaces, and Coherence for Symmetric Monoidal Bicategories. *Advances in Mathematics* 246 (Oct. 2013), 1–32. https://doi.org/10.1016/j.aim.2013.06.028

Håkon Robbestad Gylterud. 2020. Multisets in Type Theory. *Mathematical Proceedings of the Cambridge Philosophical Society* 169, 1 (July 2020), 1–18. https://doi.org/10.1017/S0305004119000045

Anthony J. G. Hey (Ed.). 1999. *Feynman and Computation: Exploring the Limits of Computers.* Perseus Books, USA.

Florent Hiver. 2021. Coq-Combi: The Coxeter Presentation of the Symmetric Group. Mathematical Components. https://github.com/math-comp/Coq-Combi/blob/1ba924ecc1a1c7714a9b3a2dbb23d91af2a1193a/theories/SymGroup/presentSn.v

Lorenz Huelsbergen. 1996. A Logically Reversible Evaluator for the Call-by-Name Lambda Calculus. *InterJournal Complex Systems* 46 (1996).

Gérard Huet. 1980. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems: Abstract Properties and Applications to Term Rewriting Systems. *J. ACM* 27, 4 (Oct. 1980), 797–821. https://doi.org/10.1145/322217.322230

J. M. E. Hyland. 2017. Classical Lambda Calculus in Modern Dress. *Mathematical Structures in Computer Science* 27, 5 (June 2017), 762–781. https://doi.org/10.1017/S0960129515000377

Martin Hyland and John Power. 2004. Symmetric Monoidal Sketches and Categories of Wirings. *Electronic Notes in Theoretical Computer Science* 100 (Oct. 2004), 31–46. https://doi.org/10.1016/j.entcs.2004.09.004

Roshan P. James and Amr Sabry. 2012. Information Effects. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/2103656.2103667

Roshan P. James and Amr Sabry. 2014. Theseus: A High-Level Language for Reversible Computation. In *Reversible Computation*.

A. Joyal and R. Street. 1993. Braided Tensor Categories. *Advances in Mathematics* 102, 1 (Nov. 1993), 20–78. https://doi.org/10.1006/aima.1993.1055

Chris Kapulkin, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. 2012. Univalence in Simplicial Sets. (March 2012). https://arxiv.org/abs/1203.2553v1

Krzysztof Kapulkin and Peter LeFanu Lumsdaine. 2021. The Simplicial Model of Univalent Foundations (after Voevodsky). *Journal of the European Mathematical Society* 23, 6 (March 2021), 2071–2126. https://doi.org/10.4171/JEMS/1050

G. M. Kelly. 1974. Coherence Theorems for Lax Algebras and for Distributive Laws. In *Category Seminar (Lecture Notes in Mathematics)*, Gregory M. Kelly (Ed.). Springer, Berlin, Heidelberg, 281–375. https://doi.org/10.1007/BFb0063106

Werner Kluge. 2000. A Reversible SE(M)CD Machine. In *Implementation of Functional Languages (Lecture Notes in Computer Science)*, Pieter Koopman and Chris Clack (Eds.). Springer, Berlin, Heidelberg, 95–113. https://doi.org/10.1007/10722298_6

Donald Ervin Knuth. 1997. *The Art of Computer Programming* (3rd ed ed.). Addison-Wesley, Reading, Mass.

DONALD E. Knuth and PETER B. Bendix. 1970. Simple Word Problems in Universal Algebras††The Work Reported in This Paper Was Supported in Part by the U.S. Office of Naval Research. In *Computational Problems in Abstract Algebra*, JOHN Leech (Ed.). Pergamon, 263–297. https://doi.org/10.1016/B978-0-08-012975-4.50028-X

Nicolai Kraus and Jakob von Raumer. 2020. Coherence via Well-Foundedness: Taming Set-Quotients in Homotopy Type Theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, Saarbrücken Germany, 662–675. https://doi.org/10.1145/3373718.3394800

Paul Krogmeier, Umang Mathur, Adithya Murali, P. Madhusudan, and Mahesh Viswanathan. 2020. Decidable Synthesis of Programs with Uninterpreted Functions. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 634–657. https://doi.org/10.1007/978-3-030-53291-8_32

Yves Lafont. 2003. Towards an Algebraic Theory of Boolean Circuits. *Journal of Pure and Applied Algebra* 184, 2 (Nov. 2003), 257–310. https://doi.org/10.1016/S0022-4049(03)00069-0

C.-A. Laisant. 1888. Sur la numération factorielle, application aux permutations. *Bulletin de la Société Mathématique de France* 2 (1888), 176–183. https://doi.org/10.24033/bsmf.378

R. Landauer. 1961. Irreversibility and Heat Generation in the Computing Process. *IBM Journal of Research and Development* 5, 3 (July 1961), 183–191. https://doi.org/10.1147/rd.53.0183

Miguel L. Laplaza. 1972. Coherence for Distributivity. In *Coherence in Categories (Lecture Notes in Mathematics)*, G. M. Kelly, M. Laplaza, G. Lewis, and Saunders Mac Lane (Eds.). Springer, Berlin, Heidelberg, 29–65. https://doi.org/10.1007/BFb0059555

F. William Lawvere. 1963. FUNCTORIAL SEMANTICS OF ALGEBRAIC THEORIES. *Proceedings of the National Academy of Sciences* 50, 5 (Nov. 1963), 869–872. https://doi.org/10.1073/pnas.50.5.869

D. H. Lehmer. 1960. Teaching Combinatorial Tricks to a Computer. In *Proceedings of Symposia in Applied Mathematics*, Richard Bellman and Marshall Hall (Eds.). Vol. 10. American Mathematical Society, Providence, Rhode Island, 179–193. https://doi.org/10.1090/psapm/010/0113289

Saunders MacLane. 1963. Natural Associativity and Commutativity. *Rice Institute Pamphlet - Rice University Studies* 49, 4 (Oct. 1963). https://scholarship.rice.edu/handle/1911/62865

Dmitri Maslov. 2003. *Reversible Logic Synthesis.* Technical Report.

Hideya Matsumoto. 1964. Générateurs et Relations Des Groupes de Weyl Généralisés. *COMPTES RENDUS HEBDOMADAIRES DES SEANCES DE L ACADEMIE DES SCIENCES* 258, 13 (1964), 3419.

Paul-André Melliès. 2019. Template Games and Differential Linear Logic. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. https://doi.org/10.1109/LICS.2019.8785830

D.M. Miller, D. Maslov, and G.W. Dueck. 2003. A Transformation Based Algorithm for Reversible Logic Synthesis. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No.03CH37451)*. 318–323. https://doi.org/10.1145/775832.775915

Martin Molzer. 2021. Cubical Agda: Simple Application of Fin: Lehmer Codes. Agda Github Community. https://github.com/agda/cubical/blob/a1d2bb38c0794f3cb00610cd6061cf9b5410518d/Cubical/Data/Fin/LehmerCode.agda

Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi. 2004. An Injective Language for Reversible Computation. In *Mathematics of Program Construction (Lecture Notes in Computer Science)*, Dexter Kozen (Ed.). Springer, Berlin, Heidelberg, 289–313. https://doi.org/10.1007/978-3-540-27764-4_16

Ian Orton and Andrew M. Pitts. 2018. Decomposing the Univalence Axiom. In *23rd International Conference on Types for Proofs and Programs (TYPES 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 104)*, Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 6:1–6:19. https://doi.org/10.4230/LIPIcs.TYPES.2017.6

Asher Peres. 1985. Reversible Logic and Quantum Computers. *Physical Review A* 32, 6 (Dec. 1985), 3266–3276. https://doi.org/10.1103/PhysRevA.32.3266

I. Petrakis. 2019. Univalent Typoids. (2019). https://www.math.lmu.de/~petrakis/Typoids.pdf

Stefano Piceghello. 2019. Coherence for Symmetric Monoidal Groupoids in HoTT/UF. (2019), 2. http://www.ii.uib.no/~bezem/abstracts/TYPES_2019_paper_10

Stefano Piceghello. 2020. Coherence for Monoidal Groupoids in HoTT. In *25th International Conference on Types for Proofs and Programs (TYPES 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 175)*, Marc Bezem and Assia Mahboubi (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 8:1–8:20. https://doi.org/10.4230/LIPIcs.TYPES.2019.8

Amr Sabry, Jacques Carette, zsparks, Chao-Hong Chen, Vikraman Choudhury, Roshan James, and movieverse. 2021. JacquesCarette/Pi-Dual: Second Alpha Release. Zenodo. https://doi.org/10.5281/zenodo.5620828

V.V. Shende, A.K. Prasad, I.L. Markov, and J.P. Hayes. 2003. Synthesis of Reversible Logic Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 22, 6 (June 2003), 710–722. https://doi.org/10.1109/TCAD.2003.811448

Zachary Sparks and Amr Sabry. 2014. Superstructural Reversible Logic. (2014), 12. https://legacy.cs.indiana.edu/~sabry/papers/reversible-logic.pdf

Arnaud Spiwack and Thierry Coquand. 2010. *Constructively Finite?* Universidad de La Rioja. 217 pages. https://hal.inria.fr/inria-00503917

Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. 2021. The Marriage of Univalence and Parametricity. *J. ACM* 68, 1 (Jan. 2021), 5:1–5:44. https://doi.org/10.1145/3429979

Tommaso Toffoli. 1980. Reversible Computing. In *Automata, Languages and Programming (Lecture Notes in Computer Science)*, Jaco de Bakker and Jan van Leeuwen (Eds.). Springer, Berlin, Heidelberg, 632–644. https://doi.org/10.1007/3-540-10003-2_104

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Univalent Foundations Program, Institute for Advanced Study. https://homotopytypetheory.org/book

Vincent Vajnovszki. 2011. A New Euler–Mahonian Constructive Bijection. *Discrete Applied Mathematics* 159, 14 (Aug. 2011), 1453–1459. https://doi.org/10.1016/j.dam.2011.05.012

André van Tonder. 2004. A Lambda Calculus for Quantum Computation. *SIAM J. Comput.* 33, 5 (Jan. 2004), 1109–1135. https://doi.org/10.1137/S0097539703432165

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proceedings of the ACM on Programming Languages* 3, ICFP (July 2019), 87:1–87:29. https://doi.org/10.1145/3341691

Noson S. Yanofsky. 2000. The syntax of coherence. *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 41, 4 (2000), 255–304. http://www.numdam.org/item/?id=CTGDC_2000__41_4_255_0

Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. 2008. Principles of a Reversible Programming Language. In *Proceedings of the 5th Conference on Computing Frontiers (CF '08)*. Association for Computing Machinery, New York, NY, USA, 43–54. https://doi.org/10.1145/1366230.1366239

Tetsuo Yokoyama and Robert Glück. 2007. A Reversible Programming Language and Its Invertible Self-Interpreter. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '07)*. Association for Computing Machinery, New York, NY, USA, 144–153. https://doi.org/10.1145/1244381.1244404

Brent Abraham Yorgey. 2014. Combinatorial Species and Labelled Structures. *Dissertations available from ProQuest* (Jan. 2014), 1–206. https://repository.upenn.edu/dissertations/AAI3668177