Computing with
Semirings and Weak Rig Groupoids

Jacques Carette and Amr Sabry

[1] McMaster University (carette@mcmaster.ca)
[2] Indiana University (sabry@indiana.edu)

Abstract. The original formulation of the Curry–Howard correspondence relates propositional logic to the simply-typed λ-calculus at three levels: the syntax of propositions corresponds to the syntax of types; the proofs of propositions correspond to programs of the corresponding types; and the normalization of proofs corresponds to the evaluation of programs. This rich correspondence has inspired our community for half a century and has been generalized to deal with more advanced logics and programming models. We propose a variant of this correspondence which is inspired by conservation of information and recent homotopy theoretic approaches to type theory.
Our proposed correspondence naturally relates semirings to reversible programming languages: the syntax of semiring elements corresponds to the syntax of types; the proofs of semiring identities correspond to (reversible) programs of the corresponding types; and equivalences between algebraic proofs correspond to meaning-preserving program transformations and optimizations. These latter equivalences are not ad hoc: the same way semirings arise naturally out of the structure of types, a categorical look at the structure of proof terms gives rise to (at least) a weak rig groupoid structure, and the coherence laws are exactly the program transformations we seek. Thus it is algebra, rather than logic, which finally leads us to our correspondence.

1 Introduction

Elementary building blocks of type theory include the empty type ($\bot$), the unit type ($\top$), the sum type ($\uplus$), and the product type ($\times$). The traditional Curry–Howard correspondence which goes back to at least 1969 relates these types to logical propositions as follows: the type $\bot$ corresponds to the absurd proposition with no proof; the type $\top$ corresponds to the trivially true proposition; the type $\tau_1 \uplus \tau_2$ corresponds to the disjunction of the corresponding propositions, and the type $\tau_1 \times \tau_2$ corresponds to the conjunction of the corresponding propositions. The following tautologies of propositional logic therefore give rise to functions witnessing the back-and-forth transformations:

$$\begin{aligned}
\tau \uplus \tau &\Leftrightarrow \tau \\
\tau \times \tau &\Leftrightarrow \tau \\
(\tau_1 \times \tau_2) \uplus \tau_3 &\Leftrightarrow (\tau_1 \uplus \tau_3) \times (\tau_2 \uplus \tau_3)
\end{aligned}$$

This connection to logic, as inspiring as it is, only cares whether a type is inhabited or not. For example, when translated to the world of types, the second tautology above states that the type $\tau \times \tau$ is inhabited iff the type τ is. Furthermore, the proofs of the two implications give rise to two functions that produce an element from one type given an element of the other. This framework is however of no direct help if one is concerned with other, richer properties of types and their relationships. For example, type isomorphisms are an important relation between types that is more refined than mere inhabitance of types as they clearly distinguish $\tau \times \tau$ and τ.

The study of type isomorphisms became popular during at least two short periods: in the early 1990s when they were used to search large libraries [30], and in the mid 2000s when they were studied from a categorical perspective [12, 13, 11]. In the last few years, type isomorphisms became one of the central concepts in homotopy type theory (HoTT) [33], where type equivalences feature prominently. These connections exposed that there is even more interesting structure arising from type isomorphisms at higher levels. For example, let Bool abbreviate the type $\top + \top$ and consider the two isomorphisms between the type Bool and itself. One of these is the identity and the other is the twist (negation) map. These isomorphisms are themselves "not equivalent" in a sense to be formalized.

The question we therefore ask is whether there is a natural correspondence, in the style of the Curry–Howard correspondence, between types and some existing mathematical entities, which would bring forth the structure of type isomorphisms and their equivalences at higher levels. We argue that, for the case of finite types, commutative semirings and their categorification are exactly these entities. In a broader sense, such a correspondence connects computation with mathematical structures common in topology and physics, thus opening the door for deeper and more fruitful interactions among these disciplines [1]. In more detail, because physical laws obey various conservation principles (including conservation of information), every computation is, at the physical level, an equivalence that preserves information. The idea that computation, at the logical and programming level, should also be based on "equivalences" (i.e., invertible processes) was originally motivated by such physical considerations [25, 4, 32, 10, 15, 29]. More recently, the rising importance of energy conservation for both tiny mobile devices and supercomputers, the shrinking size of technology at which quantum effects become noticeable, and the potential for quantum computation and communication, are additional physical considerations adding momentum to such reversible computational models [14, 7].

Outline. The next section discusses the correspondence between semirings and types at an intuitive informal level. Sec. 3 formalizes the notions of equivalences of types and equivalences of equivalences which are the semantic building blocks for the computational side of the Curry–Howard-style correspondence we aim for. Sec. 4 introduces a reversible programming language which exactly captures type equivalences. Sec. 5 lays the categorical foundation for developing a second language that exactly captures equivalences between equivalences. Sec. 6

introduces such a language. The remaining sections put our work in perspective, point out its limitations and directions for future work, and conclude.

We note that because the issues involved are quite subtle, the paper is partly an "unformalization" of an executable `Agda 2.4.2.4` package with the global `without-K` option enabled. The code is available at `http://github.com//JacquesCarette/pi-dual/Univalence`. We also make crucial use of a substantial library of categorical structures; we forked our copy from `https://github.com/copumpkin/categories` and augmented it with definitions for Groupoid, Rig Category and Bicategory. This fork is available from `https://github.com/JacquesCarette/categories`.

2 Informal Development

We explore the main ingredients that would constitute a Curry–Howard-like correspondence between (commutative) semirings and (constructive) type theory.

2.1 Semirings

We begin with the standard definition of commutative semirings.

Definition 1. *A* commutative semiring *sometimes called a* commutative rig *(ring without negative elements) consists of a set R, two distinguished elements of R named 0 and 1, and two binary operations $+$ and $\cdot$, satisfying the following relations for any $a, b, c \in R$:*

$$\begin{aligned}
0 + a &= a && (+\text{-}unit)\\
a + b &= b + a && (+\text{-}swap)\\
a + (b + c) &= (a + b) + c && (+\text{-}assoc)\\
\\
1 \cdot a &= a && (\cdot\text{-}unit)\\
a \cdot b &= b \cdot a && (\cdot\text{-}swap)\\
a \cdot (b \cdot c) &= (a \cdot b) \cdot c && (\cdot\text{-}assoc)\\
\\
0 \cdot a &= 0 && (\cdot\text{-}0)\\
(a + b) \cdot c &= (a \cdot c) + (b \cdot c) && (\cdot\text{-}+)
\end{aligned}$$

If one were to focus on the *syntax* of the semiring elements, they would be described using the following grammar:

$$a, b ::= 0 \mid 1 \mid a + b \mid a \cdot b$$

This grammar corresponds to the grammar for the finite types in type theory:

$$\tau ::= \bot \mid \top \mid \tau_1 \uplus \tau_2 \mid \tau_1 \times \tau_2$$

We will show that this — so far — superficial correspondence scratches the surface of a beautiful correspondence of rich combinatorial structure.

2.2 Semiring Identities and Isomorphisms

Having matched the syntax of semiring elements and the syntax of types, we examine the computational counterpart of the semiring identities. When viewed from the type theory side, each semiring identity asserts that two types are "equal." For example, the identity $\cdot$-unit, i.e., $1 \cdot a = a$ asserts that the types $\top \times A$ and A are "equal." One way to express such an "equality" computationally is to exhibit two functions mediating between the two types and prove that these two functions are inverses. Specifically, we define:

$$\begin{array}{ll} \mathsf{f} \ : \ \top \times A \to A & \qquad \bar{\mathsf{f}} : A \to \top \times A \\ \mathsf{f}\ (\mathsf{tt}, x) = x & \qquad \bar{\mathsf{f}}\ x = (\mathsf{tt}, x) \end{array}$$

and prove $\mathsf{f} \circ \bar{\mathsf{f}} = \bar{\mathsf{f}} \circ \mathsf{f} = \mathsf{id}$. One could use this proof to "equate" the two types but, in our proof-relevant development, it is more appropriate to keep the identity of the types separate and speak of *isomorphisms*.

2.3 Proof Relevance

In the world of semirings, there are many proofs of $a + a = a + a$. Consider

$$\begin{array}{lll} \mathsf{pf}_1 : & a + a = a + a & \text{(because = is reflexive)} \\ \mathsf{pf}_2 : & a + a = a + a & \text{(using +-swap)} \end{array}$$

In some cases, we might not care *how* a semiring identity was proved and it might then be acceptable to treat pf_1 and pf_2 as "equal." But although these two proofs of $a + a = a + a$ look identical, they use different "justifications" and these justifications are clearly *not* "equal."

When viewed from the computational side, the situation is as follows. The first proof gives rise to one isomorphism using the self-inverse function id. The second proof gives rise to another isomorphism using another self-inverse function swap defined as:

$$\begin{array}{l} \mathsf{swap} \ : \ A \uplus B \to B \uplus A \\ \mathsf{swap}\ (\mathsf{inj}_1\ x) = \mathsf{inj}_2\ x \\ \mathsf{swap}\ (\mathsf{inj}_2\ x) = \mathsf{inj}_1\ x \end{array}$$

Now it is clear that even though both id and swap can be used to establish an isomorphism between $A \uplus A$ and itself, their actions are different. Semantically speaking, these two functions are different and no program transformation or optimization should ever identify them.

The discussion above should not however lead one to conclude that programs resulting from different proofs are always semantically different. Consider for example, the following two proofs of $(a + 0) + b = a + b$. To avoid clutter in this informal presentation, we omit the justifications that refer to the fact that = is a congruence relation:

$$\begin{array}{llll} \mathsf{pf}_3 : & (a + 0) + b & = (0 + a) + b & \text{(using +-swap)} \\ & & = a + b & \text{(using +-unit)} \end{array}$$

$$\begin{array}{llll} \mathsf{pf}_4 : & (a + 0) + b & = a + (0 + b) & \text{(using +-assoc)} \\ & & = a + b & \text{(using +-unit)} \end{array}$$

On the computational side, the proofs induce the following two isomorphisms between $(A \uplus \bot) \uplus B$ and $A \uplus B$. The first isomorphism pf_3 takes the values in $(A \uplus \bot) \uplus B$ using the composition of the following two isomorphisms:

$$\begin{array}{ll}
\mathsf{f}_1 \;:\; (A \uplus \bot) \uplus B \to (\bot \uplus A) \uplus B & \overline{\mathsf{f}_1} \;:\; (\bot \uplus A) \uplus B \to (A \uplus \bot) \uplus B \\
\mathsf{f}_1(\mathsf{inj}_1\ (\mathsf{inj}_1\ x)) = \mathsf{inj}_1\ (\mathsf{inj}_2\ x) & \overline{\mathsf{f}_1}(\mathsf{inj}_1\ (\mathsf{inj}_2\ x)) = \mathsf{inj}_1\ (\mathsf{inj}_1\ x) \\
\mathsf{f}_1(\mathsf{inj}_2\ x) = \mathsf{inj}_2\ x & \overline{\mathsf{f}_1}(\mathsf{inj}_2\ x) = \mathsf{inj}_2\ x
\end{array}$$

$$\begin{array}{ll}
\mathsf{f}_2 \;:\; (\bot \uplus A) \uplus B \to A \uplus B & \overline{\mathsf{f}_2} \;:\; A \uplus B \to (\bot \uplus A) \uplus B \\
\mathsf{f}_2(\mathsf{inj}_1\ (\mathsf{inj}_2\ x)) = \mathsf{inj}_1\ x & \overline{\mathsf{f}_2}(\mathsf{inj}_1\ x) = \mathsf{inj}_1\ (\mathsf{inj}_2\ x) \\
\mathsf{f}_2(\mathsf{inj}_2\ x) = \mathsf{inj}_2\ x & \overline{\mathsf{f}_2}(\mathsf{inj}_2\ x) = \mathsf{inj}_2\ x
\end{array}$$

We calculate that composition corresponding to pf_3 as:

$$\begin{array}{ll}
\mathsf{f}_{12} \;:\; (A \uplus \bot) \uplus B \to A \uplus B & \overline{\mathsf{f}_{12}} \;:\; A \uplus B \to (A \uplus \bot) \uplus B \\
\mathsf{f}_{12}(\mathsf{inj}_1\ (\mathsf{inj}_1\ x)) = \mathsf{inj}_1\ x & \overline{\mathsf{f}_{12}}(\mathsf{inj}_1\ x) = \mathsf{inj}_1\ (\mathsf{inj}_1\ x) \\
\mathsf{f}_{12}(\mathsf{inj}_2\ x) = \mathsf{inj}_2\ x & \overline{\mathsf{f}_{12}}(\mathsf{inj}_2\ x) = \mathsf{inj}_2\ x
\end{array}$$

We can similarly calculate the isomorphism corresponding to pf_4 and verify that it is identical to the one above.

2.4 Summary

To summarize, there is a natural computational model that emerges from viewing types as syntax for semiring elements and semiring identities as type isomorphisms. The correspondence continues further between justifications for semiring identities and valid program transformations and optimizations. There is a long way however from noticing such a correspondence to formalizing it in such a way that a well-founded reversible programming language along with its accompanying program transformations and optimizations can be naturally extracted from the algebraic semiring structure. Furthermore, the correspondence between the algebraic manipulations in semirings and program transformations is so tight that it should be possible to conveniently move back and forth between the two worlds transporting results that are evident in one domain to the other. The remainder of the paper is about such a formalization and its applications.

3 Type Equivalences and Equivalences of Equivalences

The previous section used two informal notions of equivalence: between types, corresponding to semiring identities, and between programs, corresponding to proofs of semiring identities. We make this precise.

3.1 Type Equivalences

As a first approximation, Sec. 2.2 identifies two types when there is an isomorphism between them. The next section (Sec. 2.3) however reveals that we want to reason at a higher level about equivalences of such isomorphisms. We therefore follow the HoTT approach and expose one of the functions forming the isomorphism in order to explicitly encode the precise way in which the two types are equivalent. Thus, the two equivalences between Bool and itself will be distinguished by the underlying witness of the isomorphism.

Technically our definition of type equivalence relies on quasi-inverses and homotopies defined next.[3]

Definition 2 (Homotopy). *Two functions $f, g : A \to B$ are* homotopic*, written $f \sim g$, if $\forall x : A. f(x) = g(x)$. In Agda, we write:*

```
_∼_ : ∀ {A : Set} {P : A → Set} → (f g : (x : A) → P x) → Set
_∼_  {A} f g = (x : A) → f x ≡ g x
```

where Set is the universe of Agda types.

In the HoTT world, there is a distinction between the identification of two functions $f \equiv g$, and two functions producing equal values on all inputs $f \sim g$: the two notions are traditionally identified but are only *equivalent* in the HoTT context.

Definition 3 (Quasi-inverse). *For a function $f : A \to B$, a* quasi-inverse *of f is a triple (g, α, β), consisting of a function $g : B \to A$ and two homotopies $\alpha : f \circ g \sim \mathrm{id}_B$ and $\beta : g \circ f \sim \mathrm{id}_A$. In Agda, we write:*

```
record isqinv {A : Set} {B : Set} (f : A → B) : Set where
  constructor qinv
  field
    g : B → A
    α : (f ∘ g) ∼ id
    β : (g ∘ f) ∼ id
```

The terminology "quasi-inverse" was chosen in the HoTT context as a reminder that this is a poorly-behaved notion by itself as the same function $f : A \to B$ may have multiple unequal quasi-inverses; however, up to homotopy, all quasi-inverses are equivalent. From a quasi-inverse, one can build an inverse (and vice-versa); however, in a proof-relevant setting, logical equivalence is insufficient.

Definition 4 (Equivalence of types). *Two types A and B are equivalent $A \simeq B$ if there exists a function $f : A \to B$ together with a quasi-inverse for f. In Agda, we write:*

```
_≃_ : Set → Set → Set
A ≃ B = Σ (A → B) isqinv
```

[3] For reasons beyond the scope of this paper, we do not use any of the definitions of equivalence which make it a *mere proposition*, as we want a definition which is syntactically symmetric.

It is easy to prove that homotopies (for any given function space $A \to B$) are an equivalence relation. It is also straightforward to show that $\simeq$ is an equivalence relation by defining:

$$\begin{aligned}
\mathsf{id}{\simeq} &: A \simeq A \\
\mathsf{sym}{\simeq} &: (A \simeq B) \to (B \simeq A) \\
\mathsf{trans}{\simeq} &: (A \simeq B) \to (B \simeq C) \to (A \simeq C)
\end{aligned}$$

The definition of equivalence allows us to formalize the presentation of Sec. 2.2 by proving that every commutative semiring identity is satisfied by types in the universe (Set) up to $\simeq$.

Theorem 1. *The collection of all types (Set) forms a commutative semiring (up to $\simeq$).*

Proof. As expected, the additive unit is $\bot$, the multiplicative unit is $\top$, and the two binary operations are $\uplus$ and $\times$. The relevant structure in Agda is:

```
typesCSR : CommutativeSemiring (Level.suc Level.zero) Level.zero
typesCSR = record {
  Carrier = Set ;
  _≈_ = _≃_ ;  _+_ = _⊎_ ; _*_ = _×_ ;
  0# = ⊥ ; 1# = ⊤ ;
  isCommutativeSemiring = typesIsCSR }
```

The functions, homotopies, and quasi-inverses witnessing the explicit equivalences are defined within typesIsCSR and are straightforward. For future reference, we list some of these equivalences:

$$\begin{aligned}
\mathsf{unite}_{+}{\simeq} &: (\bot \uplus A) \simeq A \\
\mathsf{unite}_{+}{'}{\simeq} &: (A \uplus \bot) \simeq A \\
\mathsf{swap}_{+}{\simeq} &: (A \uplus B) \simeq (B \uplus A) \\
\mathsf{assoc}_{+}{\simeq} &: ((A \uplus B) \uplus C) \simeq (A \uplus (B \uplus C)) \\
\_{\uplus}{\simeq}\_ &: (A \simeq C) \to (B \simeq D) \to ((A \uplus B) \simeq (C \uplus D))
\end{aligned}$$

3.2 Equivalences of Equivalences

In the terminology of Sec. 2.3, an equivalence $\simeq$ denotes a proof of a semiring identity. Thus the proofs pf_1, pf_2, pf_3, and pf_4 can be written formally as:

```
pf₁ pf₂ : {A : Set} → (A ⊎ A) ≃ (A ⊎ A)
pf₁ = id≃
pf₂ = swap₊≃

pf₃ pf₄ : {A B : Set} → ((A ⊎ ⊥) ⊎ B) ≃ (A ⊎ B)
pf₃ = trans≃ (swap₊≃ ⊎≃ id≃) (unite₊≃ ⊎≃ id≃)
pf₄ = trans≃ assoc₊≃ (id≃ ⊎≃ unite₊≃)
```

In order to argue that pf$_3$ and pf$_4$ are equivalent, we therefore need a notion of equivalence of equivalences. To motivate our definition below, we first consider the obvious idea of using $\simeq$ to relate equivalences. In that case, an equivalence of equivalences of type $(A \simeq B) \simeq (A \simeq B)$ would include functions f and g mapping between $(A \simeq B)$ and itself in addition to two homotopies α and β witnessing $(f \circ g) \sim \mathsf{id}$ and $(g \circ f) \sim \mathsf{id}$ respectively. Expanding the definition of a homotopy, we note that α and β would therefore attempt to compare equivalences (which include functions) using propositional equality $\equiv$. In other words, we need to resolve to homotopies again to compare these functions: two equivalences are equivalent if there exist homotopies between their underlying functions.[4]

Definition 5 (Equivalence of equivalences). *Two equivalences $eq_1, eq_2 : A \simeq B$ are themselves equivalent, written $eq_2 \approxeq eq_2$, if $eq_1.f \sim eq_2.f$ and $eq_1.g \sim eq_2.g$. In Agda, we write:*

```
record _≋_ {A B : Set} (eq₁ eq₂ : A ≃ B) : Set where
  constructor eq
  open isqinv
  field
    f≡ : proj₁ eq₁ ∼ proj₁ eq₂
    g≡ : g (proj₂ eq₁) ∼ g (proj₂ eq₂)
```

We could now verify that indeed pf$_3$ $\approxeq$ pf$_4$. Such a proof exists in the accompanying code but requires a surprising amount of tedious infrastructure to present. We will have to wait until Secs. 5.1 and 6.4 to see this proof.

4 Programming with Equivalences

We have established and formalized a correspondence between semirings and types which relates semiring identities to the type equivalences of Def. 4. We have further introduced the infrastructure needed to reason about equivalences of equivalences so that we can reason about the relation between different proofs of the same semiring identity. As we aim to refine these relationships to a Curry–Howard-like correspondence, we now turn our attention to developing an actual programming language. The first step will be to introduce syntax that denotes type equivalences. Thus instead of having to repeatedly introduce functions and their inverses and proofs of homotopies, we will simply use a term language that exactly expresses type equivalences and nothing else.

4.1 Syntax of *Π*

In previous work, Bowman, James and Sabry [6, 20] introduced the Π family of reversible languages whose only computations are isomorphisms between types. The simplest member of Π is exactly the language we seek for capturing type

[4] Strictly speaking, the g≡ component is redundant, from a logical perspective, as it is derivable. From a computational perspective, it is very convenient.

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

Fig. 1. Π-terms [6, 20].

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

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

Fig. 2. Π-combinators.

equivalences arising from semiring identities. The syntactic components of our language are as follows:

$$
\begin{array}{ll}
(\mathit{Types}) & \tau ::= 0 \mid 1 \mid \tau_1 + \tau_2 \mid \tau_1 * \tau_2 \\
(\mathit{Values}) & v ::= () \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid (v_1, v_2) \\
(\mathit{Combinator\ types}) & \quad \tau_1 \leftrightarrow \tau_2 \\
(\mathit{Terms\ and\ Combinators}) & c ::= [\mathit{see\ Figs.\ 1\ and\ \ 2}]
\end{array}
$$

The values classified by the finite types are the conventional ones: $()$ of type 1, $(\mathsf{inl}\ v)$ and $(\mathsf{inr}\ v)$ for injections into sum types, and (v_1, v_2) for product types.

Fig. 1 gives the terms which correspond to the identities of commutative semirings. Each line of the figure introduces a pair of dual constants (where $\mathit{id}{\leftrightarrow}$, swap_{+} and swap_{*} are self-dual) that witness the type isomorphism in the middle. Fig. 2 adds to that 3 combinators $\odot$, $\oplus$, and $\otimes$, which come from the requirement that $\leftrightarrow$ be transitive (giving a sequential composition operator $\odot$), and that $\leftrightarrow$ be a congruence for both $+$ and $*$ (giving a way to take sums and products of combinators using $\oplus$ and $\otimes$ respectively). This latter congruence requirement is classically invisible, but appears when being proof-relevant.

By construction, each term in the language has an inverse:

Definition 6 (Syntactic Inverse !). *Each Π-term $c : \tau_1 \leftrightarrow \tau_2$ has a syntactic inverse $!c : \tau_2 \leftrightarrow \tau_1$. We only show a few representative clauses:*

$$
\begin{array}{rcl}
!\mathit{id}{\leftrightarrow} = \mathit{id}{\leftrightarrow} & \qquad & !(c_1 \odot c_2) = !c_2 \ \odot \ !c_1 \\
!\mathit{unite}_{+}l = \mathit{uniti}_{+}l & & !(c_1 \oplus c_2) = !c_1 \ \oplus \ !c_2 \\
!\mathit{uniti}_{+}l = \mathit{unite}_{+}l & & !(c_1 \otimes c_2) = !c_1 \ \otimes \ !c_2
\end{array}
$$

4.2 Example Programs

The family of *Π* languages was previously introduced as standalone reversible programming languages. The fragment without recursive types discussed in this paper is universal for reversible boolean circuits [20]. With the addition of recursive types and trace operators [18], *Π* becomes a Turing-complete reversible language [20, 6].

We illustrate the expressiveness of *Π* with a few small programs; we begin by defining the universe of types U:

```
data U : Set where
  ZERO  : U
  ONE   : U
  PLUS  : U → U → U
  TIMES : U → U → U
```

We then encode the type of booleans, write a few simple gates like the Toffoli gate [32], and use them to write a reversible full adder [19]:

```
BOOL : U                                BOOL³ : U
BOOL  = PLUS ONE ONE                    BOOL³ = TIMES BOOL² BOOL

BOOL² : U                               NOT : BOOL ↔ BOOL
BOOL² = TIMES BOOL BOOL                 NOT = swap₊

CNOT : BOOL² ↔ BOOL²
CNOT = dist ⊙ (id↔ ⊕ (id↔ ⊗ NOT)) ⊙ factor

TOFFOLI : TIMES BOOL BOOL² ↔ TIMES BOOL BOOL²
TOFFOLI = dist ⊙ (id↔ ⊕ (id↔ ⊗ CNOT))  ⊙ factor

PERES : BOOL³ ↔ BOOL³
PERES = (id↔ ⊗ NOT) ⊙ assocr⋆ ⊙ (id↔ ⊗ swap⋆) ⊙ TOFFOLI ⊙
  (id↔ ⊗ (NOT ⊗ id↔)) ⊙ TOFFOLI ⊙ (id↔ ⊗ swap⋆) ⊙
  (id↔ ⊗ (NOT ⊗ id↔)) ⊙ TOFFOLI ⊙ (id↔ ⊗ (NOT ⊗ id↔)) ⊙ assocl⋆

- Input: (z, ((n1, n2), cin)))
- Output: (g1, (g2, (sum, cout)))
F_ADDER : TIMES BOOL BOOL³ ↔ TIMES BOOL (TIMES BOOL BOOL²)
F_ADDER = swap⋆ ⊙ (swap⋆ ⊗ id↔) ⊙ assocr⋆ ⊙ swap⋆ ⊙ (PERES ⊗ id↔) ⊙
  assocr⋆ ⊙ (id↔ ⊗ swap⋆) ⊙ assocr⋆ ⊙ (id↔ ⊗ assocl⋆) ⊙
  (id↔ ⊗ PERES) ⊙ (id↔ ⊗ assocr⋆)
```

Although writing circuits using the raw syntax for combinators is tedious, the examples illustrate the programming language nature of *Π*. In other work, one can find a compiler from a conventional functional language to circuits [20], a systematic technique to translate abstract machines to *Π* [21], and a Haskell-like surface language [22] which can ease writing circuits. All that reinforces the first part of the title, i.e., that we can really compute with semirings.

4.3 Example Proofs

In addition to being a reversible programming language, Π is also a language for expressing proofs that correspond to semiring identities. Thus we can write variants of our proofs pf$_1$, pf$_2$, pf$_3$, and pf$_4$ from Sec. 2:

```
pf₁π pf₂π : {A : U} → PLUS A A ↔ PLUS A A
pf₁π = id↔
pf₂π = swap₊

pf₃π pf₄π : {A B : U} → PLUS (PLUS A ZERO) B ↔ PLUS A B
pf₃π = (swap₊ ⊕ id↔) ⊙ (unite₊l ⊕ id↔)
pf₄π = assocr₊ ⊙ (id↔ ⊕ unite₊l)
```

4.4 Semantics

We define the denotational semantics of Π to be type equivalences:

```
⟦_⟧ : U → Set
⟦ ZERO ⟧        = ⊥
⟦ ONE ⟧         = ⊤
⟦ PLUS t₁ t₂ ⟧  = ⟦ t₁ ⟧ ⊎ ⟦ t₂ ⟧
⟦ TIMES t₁ t₂ ⟧ = ⟦ t₁ ⟧ × ⟦ t₂ ⟧

c2equiv : {t₁ t₂ : U} → (c : t₁ ↔ t₂) → ⟦ t₁ ⟧ ≃ ⟦ t₂ ⟧
```

The function $[\![\cdot]\!]$ maps each type constructor to its Agda denotation. The function c2equiv confirms that every Π term encodes a type equivalence.

In previous work, we also defined an operational semantics for Π via forward and backward evaluators with the following signatures:

```
eval    : {t₁ t₂ : U} → (t₁ ↔ t₂) → ⟦ t₁ ⟧ → ⟦ t₂ ⟧
evalB   : {t₁ t₂ : U} → (t₁ ↔ t₂) → ⟦ t₂ ⟧ → ⟦ t₁ ⟧
```

This operational semantics serves as an adequate semantic specification if one focuses solely on a programming language for reversible boolean circuits. It is straightforward to prove that eval and evalB are inverses of each other.

If, in addition, one is also interested in using Π for expressing semiring identities as type equivalences then the following properties are of more interest:

```
lemma0 :  {t₁ t₂ : U} → (c : t₁ ↔ t₂) → (v : ⟦ t₁ ⟧) →
          eval c v ≡ proj₁ (c2equiv c) v

lemma1 :  {t₁ t₂ : U} → (c : t₁ ↔ t₂) → (v : ⟦ t₂ ⟧) →
          evalB c v ≡ proj₁ (sym≃ (c2equiv c)) v
```

The two lemmas confirm that these type equivalences are coherent with respect to the operational semantics, i.e., that the operational and denotational semantics of Π coincide.

5 Categorification

We have seen two important ways of modeling equivalences between types: using back-and-forth functions that compose to the identity (Def. 4) and using a programming language tailored to only express isomorphisms between types (Sec. 4.1). In terms of our desired Curry–Howard-like correspondence, we have so far related the syntax of semiring elements to types and the proofs of semiring identities to programs of the appropriate types. The last important component of the Curry-Howard correspondence is to relate semiring proof transformations to program transformations.

We thus need to reason about equivalences of equivalences. Attempting to discover these when working directly with equivalences, or with the syntax of a programming language, proves quite awkward. It, however, turns out that the solution to this problem is evident if we first generalize our models of type equivalences to the categorical setting. As we explain, in the right class of categories, the objects represent types, the morphisms represent type equivalences, and the *coherence conditions* will represent equivalences of equivalences. Our task of modeling equivalences of equivalences then reduces to "reading off" the coherence conditions for each instance of the general categorical framework.

5.1 Monoidal Categories

As the details matter, we will be explicit about the definition of all the categorical notions involved. We begin with the conventional definitions for monoidal and symmetric monoidal categories.

Definition 7 (Monoidal Category). *A* monoidal category *[27] is a category with the following additional structure:*

- *a bifunctor $\otimes$ called the monoidal or tensor product,*
- *an object I called the unit object, and*
- *natural isomorphisms $\alpha_{A,B,C} : (A \otimes B) \otimes C \overset{\sim}{\to} A \otimes (B \otimes C)$, $\lambda_A : I \otimes A \overset{\sim}{\to} A$, and $\rho_A : A \otimes I \overset{\sim}{\to} A$, such that the two diagrams below (known as the* associativity pentagon *and the* triangle for unit*) commute.*

$$\begin{tikzcd}
((A \otimes B) \otimes C) \otimes D \arrow[rr, "\alpha"] \arrow[d, "\alpha \otimes \mathrm{id}_D"'] & & (A \otimes B) \otimes (C \otimes D) \arrow[d, "\alpha"] \\
(A \otimes (B \otimes C)) \otimes D \arrow[dr, "\alpha"'] & & A \otimes (B \otimes (C \otimes D)) \\
& A \otimes ((B \otimes C) \otimes D) \arrow[ur, leftarrow, "\mathrm{id}_A \otimes \alpha"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
(A \otimes I) \otimes B \arrow[rr, "\alpha"] \arrow[dr, "\rho_A \otimes \mathrm{id}_B"'] & & A \otimes (I \otimes B) \arrow[dl, "\mathrm{id}_A \otimes \lambda_B"] \\
& A \otimes B &
\end{tikzcd}$$

Definition 8 (Braided and Symmetric Monoidal Categories). *A monoidal category is* braided *if it has an isomorphism* $\sigma_{A,B} : A \otimes B \tilde{\to} B \otimes A$ *where* σ *is a natural transformation which satisfies the* unit coherence *triangle (below on the left) and the* bilinearity *hexagon below. A braided monoidal category is* symmetric *if it additionally satisfies the* symmetry *triangle (below on the right).*

$$\begin{tikzcd}
& A \otimes I \arrow[dl, "\sigma"'] \arrow[dr, "\rho_A"] & \\
I \otimes A \arrow[rr, "\lambda_A"'] & & A
\end{tikzcd}
\qquad
\begin{tikzcd}
& A \otimes B \arrow[dl, "\sigma"'] \arrow[dr, "\mathrm{id}_A \otimes \mathrm{id}_B"] & \\
B \otimes A \arrow[rr, "\sigma"'] & & A \otimes B
\end{tikzcd}$$

$$\begin{tikzcd}
& A \otimes (B \otimes C) \arrow[dl, "\alpha"'] \arrow[dr, "\sigma"] & \\
(A \otimes B) \otimes C \arrow[d, "\sigma \otimes \mathrm{id}_C"'] & & (B \otimes C) \otimes A \arrow[d, "\alpha"] \\
(B \otimes A) \otimes C \arrow[dr, "\alpha"'] & & B \otimes (C \otimes A) \arrow[dl, "\mathrm{id}_B \otimes \sigma"] \\
& B \otimes (A \otimes C) &
\end{tikzcd}$$

According to Mac Lane's coherence theorem, the triangle and pentagon coherence laws for monoidal categories are justified by the desire to equate any two isomorphisms built using σ, λ, and ρ and having the same source and target. Similar considerations justify the coherence laws for symmetric monoidal categories. It is important to note that the coherence conditions do *not* imply that every pair of parallel morphisms with the same source and target are equal. Indeed, as Dosen and Petric explain:

> In Mac Lane's second coherence result of [...], which has to do with symmetric monoidal categories, it is not intended that all equations between arrows of the same type should hold. What Mac Lane does can be described in logical terms in the following manner. On the one hand, he has an axiomatization, and, on the other hand, he has a model category where arrows are permutations; then he shows that his axiomatization is complete with respect to this model. It is no wonder that his coherence problem reduces to the completeness problem for the usual axiomatization of symmetric groups [9].

From a different perspective, Baez and Dolan [2] explain the source of these coherence laws as arising from homotopy theory. In this theory, laws are only imposed up to homotopy, with these homotopies satisfying certain laws, again only up to homotopy, with these higher homotopies satisfying their own higher coherence laws, and so on. Remarkably, they report, among other results, that the pentagon identity arises when studying the algebraic structure possessed by a space that is homotopy equivalent to a loop space and that the hexagon identity arises in the context of spaces homotopy equivalent to double loop spaces.

As a concrete example relating homotopies and coherence conditions, the homotopy between pf_3 and pf_4 discussed in Sec. 3.2 follows from the coherence conditions of symmetric monoidal categories as follows:

$$\begin{aligned}
\mathsf{pf}_3 &= \mathsf{trans{\simeq}}\ (\mathsf{swap_+{\simeq}} \uplus{\simeq}\ \mathsf{id{\simeq}})\ (\mathsf{unite_+{\simeq}} \uplus{\simeq}\ \mathsf{id{\simeq}}) && \\
&\approx (\mathsf{trans{\simeq}}\ \mathsf{swap_+{\simeq}}\ \mathsf{unite_+{\simeq}}) \uplus{\simeq}\ \mathsf{id{\simeq}} && (\uplus{\simeq} \text{ is a functor}) \\
&\approx \mathsf{unite_+{}'{\simeq}} \uplus{\simeq}\ \mathsf{id{\simeq}} && (\text{unit coherence law}) \\
&\approx \mathsf{trans{\simeq}}\ \mathsf{assoc_+{\simeq}}\ (\mathsf{id{\simeq}} \uplus{\simeq}\ \mathsf{unite_+{\simeq}}) && (\text{triangle}) \\
&= \mathsf{pf}_4 &&
\end{aligned}$$

The derivation assumes that the category of types and equivalences is symmetric monoidal — a result which will be proved in a more general form in Thm. 2.

5.2 Weak Symmetric Rig Groupoids

Symmetric monoidal categories are the categorification of commutative monoids. The categorification of a commutative semiring is called a *symmetric rig category*. It is built from a *symmetric bimonoidal category* to which distributivity and absorption natural isomorphisms are added, and accompanying coherence laws. Since we can set things up so that every morphism is an isomorphism, it will also be a groupoid. Also, as the laws of the category only hold up to a higher equivalence, the entire setting is that of weak categories (aka bicategories).

There are several equivalent definitions of rig categories; we use the following from the nLab [28].

Definition 9 (Rig Category). *A* rig category *C is a category with a symmetric monoidal structure $(C, \oplus, 0)$ for addition and a monoidal structure $(C, \otimes, 1)$ for multiplication together with left and right distributivity natural isomorphisms:*

$$\begin{aligned}
d_\ell &: x \otimes (y \oplus z) \overset{\sim}{\to} (x \otimes y) \oplus (x \otimes z) \\
d_r &: (x \oplus y) \otimes z \overset{\sim}{\to} (x \otimes z) \oplus (y \otimes z)
\end{aligned}$$

and absorption/annihilation isomorphisms $a_\ell : x \otimes 0 \overset{\sim}{\to} 0$ and $a_r : 0 \otimes x \overset{\sim}{\to} 0$ satisfying coherence conditions [26] discussed below.

Definition 10 (Symmetric Rig Category). *A* symmetric rig category *is a rig category in which the multiplicative structure is symmetric.*

Definition 11 (Symmetric Rig Groupoid). *A* symmetric rig groupoid *is a symmetric rig category in which every morphism is invertible.*

The coherence conditions for rig categories were worked out by Laplaza [26]. Pages 31-35 of his paper report 24 coherence conditions numbered I to XXIV that vary from simple diagrams to quite complicated ones including a diagram with 9 nodes showing that two distinct ways of simplifying $(A \oplus B) \otimes (C \oplus D)$ to $(((A \otimes C) \oplus (B \otimes C)) \oplus (A \otimes D)) \oplus (B \otimes D)$ commute. The 24 coherence conditions are however not independent and it is sufficient to verify one of various smaller subsets, to be chosen depending on the situation. Generally speaking, the coherence laws appear rather obscure but they can be unpacked and "unformalized" to relatively understandable statements. They all express that two different means of getting between two equivalent types are equivalent. Thus we can give programming-oriented descriptions of these along the following lines:

I given $A \otimes (B \oplus C)$, swapping B and C then distributing (on the left) is the same as first distributing, then swapping the two summands;

II given $(A \oplus B) \otimes C$, first switching the order of the products then distributing (on the left) is the same as distributing (on the right) and then switching the order of both products;

IX given $(A \oplus B) \otimes (C \oplus D)$, we can either first distribute on the left, map right-distribution and finally associate, or we can go "the long way around" by right-distributing first, then mapping left-distribution, and then a long chain of administrative shuffles to get to the same point;

and so on.

Going through the details of the proof of the coherence theorem in [26] with a "modern" eye, one cannot help but think of Knuth-Bendix completion. Although it is known that the coherence laws for some categorical structures can be systematically derived in this way [3], it is also known that in the presence of certain structures (such as symmetry), Knuth-Bendix completion will not terminate. It would be interesting to know if there is indeed a systematic way to obtain these laws from the rewriting perspective but, as far as we know, there are no published results to that effect. The connections to homotopy theory cited by Baez and Dolan [2] (and mentioned in the previous section) appear to be the best hope for a rational reconstruction of the coherence laws.

5.3 The Symmetric Rig Groupoid of Types and Type Equivalences

We are now ready for the generalization of our model of types and type equivalences to a symmetric rig weak groupoid and this will, by construction, prove all equivalences between type equivalences like $\mathsf{pf}_3 \approx \mathsf{pf}_4$ that should be equated, while, again by construction, *not* identifying type equivalences like pf_1 and pf_2 that should not be equated.

Theorem 2. *The category whose objects are Agda types and whose morphisms are type equivalences is a symmetric rig groupoid.*

Proof. The definition of Category that we use is parametrized by an equivalence relation for its collection of morphisms between objects. Since we want a category with equivalences as morphisms, we naturally use $\approx$ for that notion of morphism-equality. These morphisms directly satisfy the axioms stated in the definitions of the various categories. The bulk of the work is in ensuring that the coherence conditions are satisfied up to homotopy. We only show the proof of one coherence condition, the first one in Laplaza's paper:

$$\begin{tikzcd}
A \otimes (B \oplus C) \arrow[r, "distl"] \arrow[d, "\mathrm{id}_A \otimes swap_+"'] & (A \otimes B) \oplus (A \otimes C) \arrow[d, "swap_+"] \\
A \otimes (C \oplus B) \arrow[r, "distl"'] & (A \otimes C) \oplus (A \otimes B)
\end{tikzcd}$$

We first have a lemma that shows that the two paths starting from the top left node are equivalent:

```
A×[B⊎C]→[A×C]⊎[A×B] : {A B C : Set} →
    (TE.distl ∘ (id {A = A} ×→ TE.swap₊ {B} {C})) ∼ (TE.swap₊ ∘ TE.distl)
A×[B⊎C]→[A×C]⊎[A×B] (x , inj₁ y) = refl
A×[B⊎C]→[A×C]⊎[A×B] (x , inj₂ y) = refl
```

The lemma asserts the that the two paths between $A \otimes (B \oplus C)$ and $(A \otimes C) \oplus (A \otimes B)$ are homotopic. To show that we have a groupoid, we also need to know that the converse lemma also holds, i.e. that reversing all arrows also gives a diagram for a homotopy, in other words:

```
[A×C]⊎[A×B]→A×[B⊎C] : {A B C : Set} →
    ((id ×→ TE.swap₊) ∘ TE.factorl) ∼ (TE.factorl ∘ TE.swap₊ {A × C} {A × B})
[A×C]⊎[A×B]→A×[B⊎C] (inj₁ x) = refl
[A×C]⊎[A×B]→A×[B⊎C] (inj₂ y) = refl
```

Finally we show that the forward equivalence and the backward equivalence are indeed related to the same diagram:

```
laplazaI = eq A×[B⊎C]→[A×C]⊎[A×B] [A×C]⊎[A×B]→A×[B⊎C]
```

where eq is the constructor for ≋. □

6 Programming with Equivalences of Equivalences

Following the lead of Sec. 4, we now develop an actual programming language whose terms denote equivalences of equivalences. Since we already have Π whose terms denote equivalences, what we actually need is a language whose terms denote equivalences of Π terms. One can think of such a language as a language for expressing valid program transformations and optimizations of Π programs. We will call the terms and combinators of the original Π language, level-0 terms, and the terms and combinators of the new language, level-1 terms.

As explained in the previous section, there is a systematic way to "discover" the level-1 terms which is driven by the coherence conditions. During our proofs, we collected all the level-1 terms that were needed to realize all the coherence conditions. This exercise suggested a refactoring of the original level-0 terms and a few iterations.

Let $c_1 : t_1 \leftrightarrow t_2$, $c_2 : t_2 \leftrightarrow t_3$, and $c_3 : t_3 \leftrightarrow t_4$:

$$\begin{gathered}
c_1 \odot (c_2 \odot c_3) \Leftrightarrow (c_1 \odot c_2) \odot c_3 \\
(c_1 \oplus (c_2 \oplus c_3)) \odot \mathit{assocl}_{+} \Leftrightarrow \mathit{assocl}_{+} \odot ((c_1 \oplus c_2) \oplus c_3) \\
(c_1 \otimes (c_2 \otimes c_3)) \odot \mathit{assocl}_{*} \Leftrightarrow \mathit{assocl}_{*} \odot ((c_1 \otimes c_2) \otimes c_3) \\
((c_1 \oplus c_2) \oplus c_3) \odot \mathit{assocr}_{+} \Leftrightarrow \mathit{assocr}_{+} \odot (c_1 \oplus (c_2 \oplus c_3)) \\
((c_1 \otimes c_2) \otimes c_3) \odot \mathit{assocr}_{*} \Leftrightarrow \mathit{assocr}_{*} \odot (c_1 \otimes (c_2 \otimes c_3)) \\
\mathit{assocr}_{+} \odot \mathit{assocr}_{+} \Leftrightarrow ((\mathit{assocr}_{+} \oplus \mathit{id}{\leftrightarrow}) \odot \mathit{assocr}_{+}) \odot (\mathit{id}{\leftrightarrow} \oplus \mathit{assocr}_{+}) \\
\mathit{assocr}_{*} \odot \mathit{assocr}_{*} \Leftrightarrow ((\mathit{assocr}_{*} \otimes \mathit{id}{\leftrightarrow}) \odot \mathit{assocr}_{*}) \odot (\mathit{id}{\leftrightarrow} \otimes \mathit{assocr}_{*})
\end{gathered}$$

Fig. 3. Signatures of level-1 Π-combinators: associativity

Let $a : t_1 \leftrightarrow t_2$, $b : t_3 \leftrightarrow t_4$, and $c : t_5 \leftrightarrow t_6$:

$$\begin{gathered}
((a \oplus b) \otimes c) \odot \mathit{dist} \Leftrightarrow \mathit{dist} \odot ((a \otimes c) \oplus (b \otimes c)) \\
(a \otimes (b \oplus c)) \odot \mathit{distl} \Leftrightarrow \mathit{distl} \odot ((a \otimes b) \oplus (a \otimes c)) \\
((a \otimes c) \oplus (b \otimes c)) \odot \mathit{factor} \Leftrightarrow \mathit{factor} \odot ((a \oplus b) \otimes c) \\
((a \otimes b) \oplus (a \otimes c)) \odot \mathit{factorl} \Leftrightarrow \mathit{factorl} \odot (a \otimes (b \oplus c))
\end{gathered}$$

Fig. 4. Signatures of level-1 Π-combinators: distributivity and factoring

6.1 Revised Syntax of Level-0 Terms

The inspiration of symmetric rig groupoids suggested a refactoring of Π with the following additional level-0 combinators:

$$\begin{array}{rcl}
\mathit{unite}_{+}r : & \tau + 0 \leftrightarrow \tau & : \mathit{uniti}_{+}r \\
\mathit{unite}_{*}r : & \tau * 1 \leftrightarrow \tau & : \mathit{uniti}_{*}r \\
\mathit{absorbl} : & \tau * 0 \leftrightarrow 0 & : \mathit{factorzr} \\
\mathit{distl} : & \tau_1 * (\tau_2 + \tau_3) \leftrightarrow (\tau_1 * \tau_2) + (\tau_1 * \tau_3) & : \mathit{factorl}
\end{array}$$

The added combinators are redundant, from an operational perspective, exactly because of the coherence conditions. They are however critical to the proofs, and in addition, they are often useful when representing circuits, leading to smaller programs with fewer redexes.

6.2 Syntax of Level-1 Terms

The big addition to Π is the level-1 combinators which are collected in Figs. 3 – 12. To avoid clutter we omit the names of the combinators (which are arbitrary) and omit some of the implicit type parameters. The reader should consult the code for full details.

Generally speaking, the level-1 combinators arise for the following reasons. About a third of the combinators come from the definition of the various natural isomorphisms $\alpha_{A,B,C}$, λ_A, ρ_A, $\sigma_{A,B}$, d_l, d_r, a_l and a_r. The first 4 natural

Let $c, c_1, c_2, c_3 : t_1 \leftrightarrow t_2$ and $c', c'' : t_3 \leftrightarrow t_4$:

$$
\begin{array}{cccc}
\mathit{id}{\leftrightarrow} \odot c \Leftrightarrow c & c \odot \mathit{id}{\leftrightarrow} \Leftrightarrow c & c \odot !c \Leftrightarrow \mathit{id}{\leftrightarrow} & !c \odot c \Leftrightarrow \mathit{id}{\leftrightarrow}
\end{array}
$$

$$
c \Leftrightarrow c \qquad \frac{c_1 \Leftrightarrow c_2 \quad c_2 \Leftrightarrow c_3}{c_1 \Leftrightarrow c_3} \qquad \frac{c_1 \Leftrightarrow c' \quad c_2 \Leftrightarrow c''}{c_1 \odot c_2 \Leftrightarrow c' \odot c''}
$$

Fig. 5. Signatures of level-1 Π-combinators: identity and composition

Let $c_0 : 0 \leftrightarrow 0$, $c_1 : 1 \leftrightarrow 1$, and $c : t_1 \leftrightarrow t_2$:

$$
\begin{array}{rl}
\mathit{unite}_{+}l \odot c \Leftrightarrow (c_0 \oplus c) \odot \mathit{unite}_{+}l & \mathit{uniti}_{+}l \odot (c_0 \oplus c) \Leftrightarrow c \odot \mathit{uniti}_{+}l \\
\mathit{unite}_{+}r \odot c \Leftrightarrow (c \oplus c_0) \odot \mathit{unite}_{+}r & \mathit{uniti}_{+}r \odot (c \oplus c_0) \Leftrightarrow c \odot \mathit{uniti}_{+}r \\
\mathit{unite}_{*}l \odot c \Leftrightarrow (c_1 \otimes c) \odot \mathit{unite}_{*}l & \mathit{uniti}_{*}l \odot (c_1 \otimes c) \Leftrightarrow c \odot \mathit{uniti}_{+}l \\
\mathit{unite}_{*}r \odot c \Leftrightarrow (c \otimes c_1) \odot \mathit{unite}_{*}r & \mathit{uniti}_{*}r \odot (c \otimes c_1) \Leftrightarrow c \odot \mathit{uniti}_{*}r \\
\multicolumn{2}{c}{\mathit{unite}_{*}l \Leftrightarrow \mathit{distl} \odot (\mathit{unite}_{*}l \oplus \mathit{unite}_{*}l)} \\
\mathit{unite}_{+}l \Leftrightarrow \mathit{swap}_{+} \odot \mathit{uniti}_{+}r & \mathit{unite}_{*}l \Leftrightarrow \mathit{swap}_{*} \odot \mathit{uniti}_{*}r
\end{array}
$$

Fig. 6. Signatures of level-1 Π-combinators: unit

isomorphisms actually occur twice, once for each of the symmetric monoidal structures at play. Each natural isomorphism is composed of 2 natural transformations (one in each direction) that must compose to the identity. This in turn induces 4 coherence laws: two *naturality laws* which indicate that the combinator commutes with structure construction, and two which express that the resulting combinators are left and right inverses of each other. We note that the mere desire that $\oplus$ be a bifunctor induces 3 coherence laws. And then of course each "structure" (monoidal, braided, symmetric) comes with more, as outlined in the previous section, culminating with 13 additional coherence laws for the rig structure.

In our presentation, we group the level-1 combinators according to the dominant property of interest, e.g., associativity in Fig. 3, or according to the main two interacting properties, e.g., commutativity and associativity in Fig. 7. It is worth noting that most (but not all) of the properties involving only $\oplus$ were already in Agda's standard library (in Data.Sum.Properties to be precise), whereas all properties involving only $\otimes$ were immediately provable due to η expansion. Nevertheless, for symmetry and clarity, we created a module Data.Prod.Properties to collect all of these. None of the mixed properties involved with distributivity and absorption were present, although the proofs for all of them were straightforward. Their statement, on the other hand, was at times rather complex (see Data.SumProd.Properties).

Let $c_1 : t_1 \leftrightarrow t_2$ and $c_2 : t_3 \leftrightarrow t_4$:

$$\begin{array}{c}
\mathit{swap}_+ \odot (c_1 \oplus c_2) \Leftrightarrow (c_2 \oplus c_1) \odot \mathit{swap}_+ \quad \mathit{swap}_* \odot (c_1 \otimes c_2) \Leftrightarrow (c_2 \otimes c_1) \odot \mathit{swap}_* \\
(\mathit{assocr}_+ \odot \mathit{swap}_+) \odot \mathit{assocr}_+ \Leftrightarrow ((\mathit{swap}_+ \oplus \mathit{id}{\leftrightarrow}) \odot \mathit{assocr}_+) \odot (\mathit{id}{\leftrightarrow} \oplus \mathit{swap}_+) \\
(\mathit{assocl}_+ \odot \mathit{swap}_+) \odot \mathit{assocl}_+ \Leftrightarrow ((\mathit{id}{\leftrightarrow} \oplus \mathit{swap}_+) \odot \mathit{assocl}_+) \odot (\mathit{swap}_+ \oplus \mathit{id}{\leftrightarrow}) \\
(\mathit{assocr}_* \odot \mathit{swap}_*) \odot \mathit{assocr}_* \Leftrightarrow ((\mathit{swap}_* \otimes \mathit{id}{\leftrightarrow}) \odot \mathit{assocr}_*) \odot (\mathit{id}{\leftrightarrow} \otimes \mathit{swap}_*) \\
(\mathit{assocl}_* \odot \mathit{swap}_*) \odot \mathit{assocl}_* \Leftrightarrow ((\mathit{id}{\leftrightarrow} \otimes \mathit{swap}_*) \odot \mathit{assocl}_*) \odot (\mathit{swap}_* \otimes \mathit{id}{\leftrightarrow})
\end{array}$$

Fig. 7. Signatures of level-1 Π-combinators: commutativity and associativity

Let $c_1 : t_1 \leftrightarrow t_2$, $c_2 : t_3 \leftrightarrow t_4$, $c_3 : t_1 \leftrightarrow t_2$, and $c_4 : t_3 \leftrightarrow t_4$.
Let $a_1 : t_5 \leftrightarrow t_1$, $a_2 : t_6 \leftrightarrow t_2$, $a_3 : t_1 \leftrightarrow t_3$, and $a_4 : t_2 \leftrightarrow t_4$.

$$\begin{array}{c}
\dfrac{c_1 \Leftrightarrow c_3 \quad c_2 \Leftrightarrow c_4}{c_1 \oplus c_2 \Leftrightarrow c_3 \oplus c_4} \qquad \dfrac{c_1 \Leftrightarrow c_3 \quad c_2 \Leftrightarrow c_4}{c_1 \otimes c_2 \Leftrightarrow c_3 \otimes c_4} \\
\mathit{id}{\leftrightarrow} \oplus \mathit{id}{\leftrightarrow} \Leftrightarrow \mathit{id}{\leftrightarrow} \qquad \mathit{id}{\leftrightarrow} \otimes \mathit{id}{\leftrightarrow} \Leftrightarrow \mathit{id}{\leftrightarrow} \\
(a_1 \odot a_3) \oplus (a_2 \odot a_4) \Leftrightarrow (a_1 \oplus a_2) \odot (a_3 \oplus a_4) \\
(a_1 \odot a_3) \otimes (a_2 \odot a_4) \Leftrightarrow (a_1 \otimes a_2) \odot (a_3 \otimes a_4)
\end{array}$$

Fig. 8. Signatures of level-1 Π-combinators: functors

6.3 Example Level-1 Programs

A pleasant outcome of having the level-1 terms is that they also give rise to an interesting programming language which, in our context, can be viewed as a language for expressing transformations and optimizations of boolean circuits. We illustrate the idea with a few small examples.

Figs. 3–12 contain rules to manipulate well-typed code fragments by rewriting them in a small-step fashion. In their textual form, the rules are certainly not intuitive. They however become "evidently correct" transformations on circuits when viewed diagrammatically. As an example, consider two arbitrary Π-combinators representing circuits of the given types:

```
c₁ : {B C : U} →  B ↔ C
c₂ : {A D : U} →  A ↔ D
```

Now consider the circuits p_1 and p_2 which use c_1 and c_2 as shown below:

```
p₁ p₂ : { A B C D : U } → PLUS A B ↔ PLUS C D
p₁ = swap₊ ⊙ ( c₁ ⊕ c₂ )
p₂ = ( c₂ ⊕ c₁ ) ⊙ swap₊
```

As reversible circuits, p_1 and p_2 evaluate as follows. If p_1 is given the value inl a, it first transforms it to inr a, and then passes it to c_2. If p_2 is given the value inl a, it first passes it to c_2 and then flips the tag of the result. Since

$$\begin{aligned}
\mathit{unite}_+r \oplus \mathit{id}{\leftrightarrow} \quad &\Leftrightarrow \quad \mathit{assocr}_+ \odot (\mathit{id}{\leftrightarrow} \oplus \mathit{unite}_+l) \\
\mathit{unite}_*r \otimes \mathit{id}{\leftrightarrow} \quad &\Leftrightarrow \quad \mathit{assocr}_* \odot (\mathit{id}{\leftrightarrow} \otimes \mathit{unite}_*l)
\end{aligned}$$

Fig. 9. Signatures of level-1 *Π*-combinators: unit and associativity

Let $c : t_1 \leftrightarrow t_2$:

$$(c \otimes \mathit{id}{\leftrightarrow}) \odot \mathit{absorbl} \Leftrightarrow \mathit{absorbl} \odot \mathit{id}{\leftrightarrow} \qquad (\mathit{id}{\leftrightarrow} \otimes c) \odot \mathit{absorbr} \Leftrightarrow \mathit{absorbr} \odot \mathit{id}{\leftrightarrow}$$

$$\mathit{id}{\leftrightarrow} \odot \mathit{factorzl} \Leftrightarrow \mathit{factorzl} \odot (\mathit{id}{\leftrightarrow} \otimes c) \qquad \mathit{id}{\leftrightarrow} \odot \mathit{factorzr} \Leftrightarrow \mathit{factorzr} \odot (c \otimes \mathit{id}{\leftrightarrow})$$

$$\mathit{absorbr} \Leftrightarrow \mathit{absorbl}$$

$$\mathit{absorbr} \Leftrightarrow (\mathit{distl} \odot (\mathit{absorbr} \oplus \mathit{absorbr})) \odot \mathit{unite}_+l$$

$$\mathit{unite}_*r \Leftrightarrow \mathit{absorbr} \qquad \mathit{absorbl} \Leftrightarrow \mathit{swap}_* \odot \mathit{absorbr}$$

$$\mathit{absorbr} \Leftrightarrow (\mathit{assocl}_* \odot (\mathit{absorbr} \otimes \mathit{id}{\leftrightarrow})) \odot \mathit{absorbr}$$

$$(\mathit{id}{\leftrightarrow} \otimes \mathit{absorbr}) \odot \mathit{absorbl} \Leftrightarrow (\mathit{assocl}_* \odot (\mathit{absorbl} \otimes \mathit{id}{\leftrightarrow})) \odot \mathit{absorbr}$$

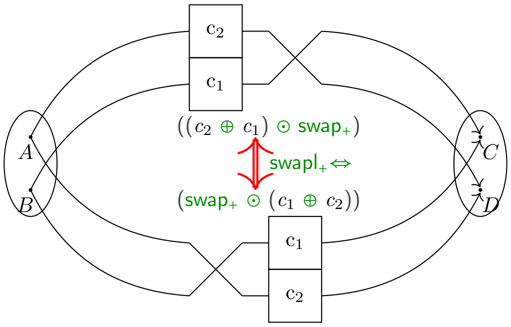
$$\mathit{id}{\leftrightarrow} \otimes \mathit{unite}_+l \Leftrightarrow (\mathit{distl} \odot (\mathit{absorbl} \oplus \mathit{id}{\leftrightarrow})) \odot \mathit{unite}_+l$$

Fig. 10. Signatures of level-1 *Π*-combinators: zero

c_2 is functorial, it must act polymorphically on its input and hence the two evaluations must produce the same result. The situation for the other possible input value is symmetric. This extensional reasoning is embedded once and for all in the proofs of coherence and distilled in a level-1 combinator (see the first combinator in Fig. 7):

$$\begin{aligned}
&\mathsf{swapl}_+\Leftrightarrow : \{t_1\ t_2\ t_3\ t_4 : \mathsf{U}\}\ \{c_1 : t_1 \leftrightarrow t_2\}\ \{c_2 : t_3 \leftrightarrow t_4\} \to \\
&\qquad (\mathsf{swap}_+ \odot (c_1 \oplus c_2)) \Leftrightarrow ((c_2 \oplus c_1) \odot \mathsf{swap}_+)
\end{aligned}$$

Categorically speaking, this combinator expresses exactly that the braiding $\sigma_{A,B}$ is a natural transformation, in other words that $\sigma_{A,B}$ must commute with $\oplus$. Pictorially, $\mathsf{swapl}_+\Leftrightarrow$ is a 2-path showing how the two programs can be transformed to one another. This can be visualized by imagining the connections as wires whose endpoints are fixed: holding the wires on the right side of the top path and flipping them produces the connection in the bottom path:



$$
\begin{aligned}
((\mathit{assocl}_{+} \otimes \mathit{id}{\leftrightarrow}) \odot \mathit{dist}) \odot (\mathit{dist} \oplus \mathit{id}{\leftrightarrow}) &\Leftrightarrow (\mathit{dist} \odot (\mathit{id}{\leftrightarrow} \oplus \mathit{dist})) \odot \mathit{assocl}_{+} \\
\mathit{assocl}_{*} \odot \mathit{distl} &\Leftrightarrow ((\mathit{id}{\leftrightarrow} \otimes \mathit{distl}) \odot \mathit{distl}) \odot (\mathit{assocl}_{*} \oplus \mathit{assocl}_{*}) \\
(\mathit{distl} \odot (\mathit{dist} \oplus \mathit{dist})) \odot \mathit{assocl}_{+} &\Leftrightarrow \mathit{dist} \odot (\mathit{distl} \oplus \mathit{distl}) \odot \mathit{assocl}_{+} \odot \\
&\quad\;\; (\mathit{assocr}_{+} \oplus \mathit{id}{\leftrightarrow}) \odot \\
&\quad\;\; ((\mathit{id}{\leftrightarrow} \oplus \mathit{swap}_{+}) \oplus \mathit{id}{\leftrightarrow}) \odot \\
&\quad\;\; (\mathit{assocl}_{+} \oplus \mathit{id}{\leftrightarrow})
\end{aligned}
$$

Fig. 11. Signatures of level-1 Π-combinators: associativity and distributivity

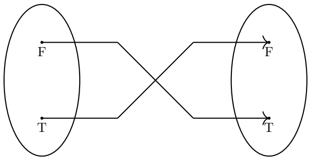
$$
\begin{aligned}
(\mathit{id}{\leftrightarrow} \otimes \mathit{swap}_{+}) \odot \mathit{distl} &\Leftrightarrow \mathit{distl} \odot \mathit{swap}_{+} \\
\mathit{dist} \odot (\mathit{swap}_{*} \oplus \mathit{swap}_{*}) &\Leftrightarrow \mathit{swap}_{*} \odot \mathit{distl}
\end{aligned}
$$

Fig. 12. Signatures of level-1 Π-combinators: commutativity and distributivity

The fact that the current syntax is far from intuitive suggests that it might be critical to have either a diagrammatic interface similar to Quantomatic [8] (which only works for traced symmetric monoidal categories) or a radically different syntactic notation such as Penrose's abstract tensor notation [23, 24].

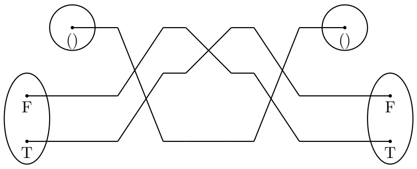
We conclude this section with a small but complete example showing how to prove the equivalence of two circuits implementing boolean negation. The first circuit uses the direct realization of boolean negation:

```
NOT₁ : BOOL ↔ BOOL
NOT₁ = Pi0.swap₊
```



The second circuit is more convoluted:

```
NOT₂ : BOOL ↔ BOOL
NOT₂ =
  uniti⋆l ⊙
  Pi0.swap⋆ ⊙
  (Pi0.swap₊ ⊗ id↔) ⊙
  Pi0.swap⋆ ⊙
  unite⋆l
```



Here is a complete proof in level-1 Π using the small-step rewriting style that shows that the two circuits are equivalent. The proofs uses the names of the level-1 combinators from the accompanying code.

```
negEx : NOT₂ ⇔ NOT₁
negEx = uniti⋆l ⊙ (Pi0.swap⋆ ⊙ ((Pi0.swap₊ ⊗ id↔) ⊙ (Pi0.swap⋆ ⊙ unite⋆l)))
    ⇔⟨ id⇔ ⊡ assoc⊙l ⟩
  uniti⋆l ⊙ ((Pi0.swap⋆ ⊙ (Pi0.swap₊ ⊗ id↔)) ⊙ (Pi0.swap⋆ ⊙ unite⋆l))
    ⇔⟨ id⇔ ⊡ (swapl⋆⇔ ⊡ id⇔) ⟩
```

```
uniti⋆l ⊙ (((id↔ ⊗ Pi0.swap₊) ⊙ Pi0.swap⋆) ⊙ (Pi0.swap⋆ ⊙ unite⋆l))
  ⇔⟨ id⇔ ⊡ assoc⊙r ⟩
uniti⋆l ⊙ ((id↔ ⊗ Pi0.swap₊) ⊙ (Pi0.swap⋆ ⊙ (Pi0.swap⋆ ⊙ unite⋆l)))
  ⇔⟨ id⇔ ⊡ (id⇔ ⊡ assoc⊙l) ⟩
uniti⋆l ⊙ ((id↔ ⊗ Pi0.swap₊) ⊙ ((Pi0.swap⋆ ⊙ Pi0.swap⋆) ⊙ unite⋆l))
  ⇔⟨ id⇔ ⊡ (id⇔ ⊡ (linv⊙l ⊡ id⇔)) ⟩
uniti⋆l ⊙ ((id↔ ⊗ Pi0.swap₊) ⊙ (id↔ ⊙ unite⋆l))
  ⇔⟨ id⇔ ⊡ (id⇔ ⊡ idl⊙l) ⟩
uniti⋆l ⊙ ((id↔ ⊗ Pi0.swap₊) ⊙ unite⋆l)
  ⇔⟨ assoc⊙l ⟩
(uniti⋆l ⊙ (id↔ ⊗ Pi0.swap₊)) ⊙ unite⋆l
  ⇔⟨ unitil⋆⇔l ⊡ id⇔ ⟩
(Pi0.swap₊ ⊙ uniti⋆l) ⊙ unite⋆l
  ⇔⟨ assoc⊙r ⟩
Pi0.swap₊ ⊙ (uniti⋆l ⊙ unite⋆l)
  ⇔⟨ id⇔ ⊡ linv⊙l ⟩
Pi0.swap₊ ⊙ id↔
  ⇔⟨ idr⊙l ⟩
Pi0.swap₊ □
```

6.4 Example Level-1 Proof

In addition to proving circuit optimizations, we can also prove equivalences of semiring proofs. As discussed in Sec. 2.3, we expect $\mathsf{pf}_3\pi$ and $\mathsf{pf}_4\pi$ to be equivalent proofs. The following derivation shows the derivation using level-1 combinators:

```
pfEx : {A B : U} → pf₃π {A} {B} ⇔ pf₄π {A} {B}
pfEx {A} {B} =
  (Pi0.swap₊ ⊕ id↔) ⊙ (unite₊l ⊕ id↔)
    ⇔⟨  hom⊙⊕⇔ ⟩
  (Pi0.swap₊ ⊙ unite₊l) ⊕ (id↔ ⊙ id↔)
    ⇔⟨ resp⊕⇔  unite₊r-coh-r idl⊙l ⟩
  unite₊r ⊕ id↔
    ⇔⟨ triangle⊕l ⟩
  Pi0.assocr₊ ⊙ (id↔ ⊕ unite₊l) □
```

6.5 Semantics

Each level-1 combinator whose signature is in Figs. 3–12 gives rise to an equivalence of equivalences of types. Furthermore, the level-1 combinators are coherent with the respect to the level-0 semantics. Formally, in Agda, we have:

```
cc2equiv :  {t₁ t₂ : U} {c₁ c₂ : t₁ ↔ t₂} (ce : c₁ ⇔ c₂) →
            PiEquiv.c2equiv c₁ ≋ PiEquiv.c2equiv c₂
```

In other words, equivalent programs exactly denote equivalent equivalences.

This is all compatible with the operational semantics as well, so that equivalent programs always give the same values; more amusingly, if we run one program then run an equivalent program backwards, we get the identity:

$$\begin{aligned}
&\mathsf{\approx\!\Rightarrow\!\equiv} : \{t_1\ t_2 : \mathsf{U}\}\ (c_1\ c_2 : t_1 \leftrightarrow t_2)\ (e : c_1 \Leftrightarrow c_2) \to \mathsf{eval}\ c_1 \sim \mathsf{eval}\ c_2 \\
&\mathsf{ping\text{-}pong} : \{t_1\ t_2 : \mathsf{U}\}\ (c_1\ c_2 : t_1 \leftrightarrow t_2)\ (e : c_1 \Leftrightarrow c_2) \to (\mathsf{evalB}\ c_2 \circ \mathsf{eval}\ c_1) \sim \mathsf{id}
\end{aligned}$$

It should be stressed that c_1 and c_2 can be arbitrarily complex programs (albeit equivalent), and still the above optimization property holds. So we have the promise of a very effective optimizer for such programs.

The next theorem is both trivial (as it holds by construction), and central to the correspondence: we distilled the level-1 combinators to make its proof trivial. It shows that the two levels of Π form a symmetric rig groupoid, thus capturing equivalences of types at level-0, and equivalences of equivalences at level-1.

Theorem 3. *The universe U and Π terms and combinators form a symmetric rig groupoid.*

Proof. The objects of the category are the syntax of finite types, and the morphisms are the Π terms and combinators. Short proofs establish that these morphisms satisfy the axioms stated in the definitions of the various categories. The bulk of the work is in ensuring that the coherence conditions are satisfied. As explained earlier in this section, this required us to add a few additional level-0 Π combinators and then to add a whole new layer of level-1 combinators witnessing enough equivalences of level-0 Π combinators to satisfy the coherence laws (see Figs. 3–12).

7 Conclusion

The traditional Curry-Howard correspondence is based on “mere logic (to use the HoTT terminology).” That is, it is based around *proof inhabitation*: two types, like two propositions, are regarded as “the same” when one is inhabited if and only if the other is. In that sense, the propositions A and $A \wedge A$, are indeed the same, as are the types T and $T \times T$. This is all centered around proof irrelevant mathematics.

What we have shown is that if we shift to proof relevant mathematics, computationally relevant equivalences, explicit homotopies, and algebra, something quite new emerges: an actual isomorphism between proof terms and reversible computations. Furthermore, what algebraic structure to use is not mysterious: it is exactly the algebraic structure of the semantics. In the case of finite types (with sums and products), this turns out to be commutative semirings.

But the Curry-Howard correspondence promises more: that proof transformations correspond to program transformations. In a proof irrelevant setting, this is rather awkward; similarly, in a extensional setting, program equivalence is a rather coarse concept. But, in our setting, both of these issues disappear. The key to proceed was to realize that there exist combinators which make equivalences look like a semiring, but do not actually have a semiring structure. The next insight is to "remember" that a monoidal category is really a model of a *typed monoid*; in a way, a monoidal category is a *categorified* monoid. So what we needed was a categorified version of commutative semirings. Luckily, this had already been done, in the form of Rig categories. Modifying this to have a weaker notion of equivalence and having all morphisms invertible was quite straightforward.

Again, being proof relevant mattered: it quickly became apparent that the *coherence laws* involved in weak Rig Groupoids were *exactly* the program equivalences that were needed. So rather than fumbling around finding various equivalences and hoping to stumble on enough of them to be complete, a systematic design emerged: given a 1-algebra of types parametrized by an equivalence $\simeq$, one should seek a 2-algebra (aka typed algebra, aka categorification of the given 1-algebra) that corresponds to it. The coherence laws then emerge as a complete set of program transformations. This, of course, clearly points the way to further generalizations.

The correspondence between rigs and types established in the paper provides a semantically well-founded approach to the representation, manipulation, and optimization of reversible circuits with the following main ingredients:

- reversible circuits are represented as terms witnessing morphisms between finite types in a symmetric rig groupoid;
- the term language for reversible circuits is universal; it could be used as a standalone point-free programming language or as a target for a higher-level language with a more conventional syntax;
- the symmetric rig groupoid structure ensures that programs can be combined using sums and products satisfying the familiar laws of these operations;
- the *weak* versions of the categories give us a second level of morphisms that relate programs to equivalent programs and is exactly captured in the coherence conditions of the categories; this level of morphisms also comes equipped with sums and products with the familiar laws and the coherence conditions capture how these operations interact with sequential composition;
- a sound and complete optimizer for reversible circuits can be represented as terms that rewrite programs in small steps witnessing this second level of morphisms.

From a much more general perspective, our result can be viewed as part of a larger programme aiming at a better integration of several disciplines most notably computation, topology, and physics. Computer science has traditionally been founded on models such as the λ-calculus which are at odds with the increasingly relevant physical principle of conservation of information as well as the recent foundational proposal of HoTT that identifies equivalences (i.e., reversible, information-preserving, functions) as a primary notion of interest.[5] Currently, these reversible functions are a secondary notion defined with reference to the full λ-calculus in what appears to be a detour. In more detail,

[5] The λ-calculus is not even suitable for keeping track of computational resources; linear logic [16] is a much better framework for that purpose but it does not go far enough as it only tracks "multiplicative resources" [31].

current constructions start with the class of all functions $A \to B$, then introduce constraints to filter those functions which correspond to type equivalences $A \simeq B$, and then attempt to look for a convenient computational framework for effective programming with type equivalences. As we have shown, in the case of finite types, this is just convoluted since the collection of functions corresponding to type equivalences is the collection of isomorphisms between finite types and these isomorphisms can be inductively defined, giving rise to a well-behaved programming language and its optimizer.

More generally, reversible computational models — in which all functions have inverses — are known to be universal computational models [4] and more importantly they can be defined without any reference to irreversible functions, which ironically become the derived notion [17]. It is, therefore, at least plausible that a variant of HoTT based exclusively on reversible functions that directly correspond to equivalences would have better computational properties. Our current result is a step, albeit preliminary, in that direction as it only applies to finite types. However, it is plausible that this categorification approach can be generalized to accommodate higher-order functions. The intuitive idea is that our current development based on the categorification of the commutative semiring of the natural numbers might be generalizable to the categorification of the ring of integers or even to the categorification of the field of rational numbers. The generalization to rings would introduce *negative types* and the generalization to fields would further introduce *fractional types*. It is even possible to conceive of more exotic types such as types with square roots and imaginary numbers by further generalizing the work to the field of *algebraic numbers*. These types have been shown to make sense in computations involving recursive datatypes such as trees that can be viewed as solutions to polynomials over type variables [5, 12, 13].

Acknowledgement.We would like sincerely thank the reviewers for their excellent and detailed comments. This material is based upon work supported by the National Science Foundation under Grant No. 1217454.

Bibliography

[1] John Baez and Mike Stay. Physics, topology, logic and computation: A Rosetta stone. arXiv:0903.0340 [quant-ph], 2009.

[2] John C. Baez and James Dolan. Categorification. In Higher Category Theory, Contemp. Math. 230, 1998, pp. 1-36., 1998.

[3] Tibor Beke. Categorification, term rewriting and the knuth–bendix procedure. *Journal of Pure and Applied Algebra*, 215(5):728 – 740, 2011.

[4] C. H. Bennett. Logical reversibility of computation. *IBM J. Res. Dev.*, 17:525–532, November 1973.

[5] A. Blass. Seven trees in one. *Journal of Pure and Applied Algebra*, 103(1-21), 1995.

[6] William J. Bowman, Roshan P. James, and Amr Sabry. Dagger traced symmetric monoidal categories and reversible programming. In *RC*, 2011.

[7] Erik P. DeBenedictis. Reversible logic for supercomputing. In *Proceedings of the 2Nd Conference on Computing Frontiers*, CF '05, pages 391–402, New York, NY, USA, 2005. ACM.

[8] Lucas Dixon and Aleks Kissinger. Open graphs and monoidal theories. arXiv:1011.4114, 2010.

[9] K. Dosen and Z. Petric. *Proof-Theoretical Coherence.* KCL Publications (College Publications), London, 2004. (revised version available at: `http://www.mi.sanu.ac.yu/~kosta/coh.pdf`).

[10] Richard Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21:467–488, 1982.

[11] M. P. Fiore, R. Di Cosmo, and V. Balat. Remarks on isomorphisms in typed calculi with empty and sum types. *Annals of Pure and Applied Logic*, 141(1-2):35–50, 2006.

[12] Marcelo Fiore. Isomorphisms of generic recursive polynomial types. In *POPL*, pages 77–88. ACM, 2004.

[13] Marcelo Fiore and Tom Leinster. An objective representation of the Gaussian integers. *Journal of Symbolic Computation*, 37(6):707 – 716, 2004.

[14] Michael P. Frank. *Reversibility for efficient computing.* PhD thesis, Massachusetts Institute of Technology, 1999.

[15] E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21(3):219–253, 1982.

[16] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[17] Alexander S. Green and Thorsten Altenkirch. From reversible to irreversible computations. *Electron. Notes Theor. Comput. Sci.*, 210:65–74, July 2008.

[18] Masahito Hasegawa. Recursion from cyclic sharing: Traced monoidal categories and models of cyclic lambda calculi. In *TLCA*, pages 196–213, 1997.

[19] Md. Saiful Islam. A novel quantum cost efficient reversible full adder gate in nanotechnology. arXiv:1008.3533, 2010.

[20] Roshan P. James and Amr Sabry. Information effects. In *POPL*, pages 73–84. ACM, 2012.

[21] Roshan P. James and Amr Sabry. Isomorphic interpreters from logically reversible abstract machines. In *RC*, 2012.

[22] Roshan P. James and Amr Sabry. Theseus: A high-level language for reversible computation. In *Reversible Computation*, 2014. Booklet of work-in-progress and short reports.

[23] Aleks Kissinger. Abstract tensor systems as monoidal categories. arXiv:1308.3586, 2013.

[24] Aleks Kissinger and David Quick. Tensors, !-graphs, and non-commutative quantum structures. arXiv:1412.8552, 2014.

[25] R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5:183–191, July 1961.

[26] MiguelL. Laplaza. Coherence for distributivity. In G.M. Kelly, M. Laplaza, G. Lewis, and Saunders Mac Lane, editors, *Coherence in Categories*, volume 281 of *Lecture Notes in Mathematics*, pages 29–65. Springer Verlag, Berlin, 1972.

[27] Saunders Mac Lane. *Categories for the working mathematician.* Springer-Verlag, 1971.

[28] nLab. rig category. `http://ncatlab.org/nlab/show/rig+category`, 2015.

[29] Asher Peres. Reversible logic and quantum computers. *Phys. Rev. A*, 32(6), Dec 1985.

[30] Mikael Rittri. Using types as search keys in function libraries. In *FPCA*, 1989.

[31] Zachary Sparks and Amr Sabry. Superstructural reversible logic. In *3rd International Workshop on Linearity*, 2014.

[32] Tommaso Toffoli. Reversible computing. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 632–644. Springer-Verlag, 1980.

[33] Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.