Strictness by Rewrite: Formalising Bipermutative Categories and Quantum Program Semantics in Agda

MALIN ALTENMÜLLER, University of Edinburgh, Scotland
ROBIN KAARSGAARD, University of Southern Denmark, Denmark

Rig categories provide a categorical foundation for the semantics of quantum programming languages, but their formalisation in Agda is hindered by the treatment of monoidal structures as *weak*: every structural equation must be stated explicitly, even when it carries no computational content. Because rig categories involve a large number of coherence conditions, formal proofs quickly become overwhelmed by structural bookkeeping.

Here, we show that Agda's rewrite rule feature can be used to overcome this problem. By promoting selected structural equivalences to definitional equalities, we effectively model semi-strict rig categories (known as bipermutative categories), allowing proofs to focus on computationally meaningful content.

We demonstrate this approach on $\sqrt{\Pi}$, a computationally universal intermediate language for unitary quantum programming equipped with a complete equational theory for standard gate sets. In Agda, proofs of identities in $\sqrt{\Pi}$ follow the same structure as their pen-and-paper counterparts—without explicit structural rearrangements.

1 Strict monoidal categories

Symmetric monoidal categories define structures in which processes can be composed both in sequence (by function composition $\fatsemi$) and in parallel (by the tensor product $\otimes$). The data of a symmetric monoidal category consists of structure isomorphisms, for example associativity and unit of the tensor product:

$$\begin{aligned} \alpha_{A,B,C} &: (A \otimes B) \otimes C \leftrightarrow A \otimes (B \otimes C), \\ \lambda_A &: 1 \otimes A \leftrightarrow A, \rho_A : A \otimes 1 \leftrightarrow A, \end{aligned} \tag{1}$$

subject to certain equations, called the *coherence conditions*. In a *strict* monoidal category, the associator and unitors of the category carry no computational content; they are all the identity natural transformation. The graphical syntax of *string diagrams* for monoidal categories encodes the strictness property by representing both sides of the isomorphisms as the same diagram. For example, both sides of the associator α in Equation 1 are expressed by the diagram with three parallel wires A, B, and C (without explicit association).

2 Rig categories

A rig category (or bimonoidal category; see [3] for a thorough treatment) consists of two monoidal structures, $\oplus$ and $\otimes$, with $\otimes$ distributing over $\oplus$, together with a number of coherence conditions. In the syntax for rig categories, we may think of objects as

Authors' Contact Information: Malin Altenmüller, University of Edinburgh, Scotland, malin.altenmuller@ed.ac.uk; Robin Kaarsgaard, University of Southern Denmark, Denmark, kaarsgaard@imada.sdu.dk.

types and morphisms as terms between types, which explains some of the names in the implementation:

```
data PiTy : Set where
  zero : PiTy
  one : PiTy
  _⊕_ : PiTy → PiTy → PiTy
  _⊗_ : PiTy → PiTy → PiTy
```

```
data _↔_ : PiTy → PiTy → Set where
  id : (A : PiTy) → A ↔ A
  sym : A ↔ B → B ↔ A
  _⨟_ : A ↔ B → B ↔ C → A ↔ C

  swap⊕ : (A B : PiTy) → (A ⊕ B) ↔ (B ⊕ A)
  swap⊗ : (A B : PiTy) → (A ⊗ B) ↔ (B ⊗ A)

  _⊕C_ : A ↔ B → C ↔ D → (A ⊕ C) ↔ (B ⊕ D)
  _⊗C_ : A ↔ B → C ↔ D → (A ⊗ C) ↔ (B ⊗ D)
```

In this category, objects are sequences of zero and one objects, connected by the two tensor products, $\otimes$ and $\oplus$. Morphisms are invertible and defined as the data type $\leftrightarrow$. In addition to an identity morphism for each object, we can form composition $\fatsemi$ of morphisms as well as compute the inverse of a morphism by using the sym constructor. Thus, the relation specifying morphisms in the category is an equivalence relation on objects. Among these, the only morphisms with computational content are the swap$\otimes$ and swap$\oplus$, which exchange the positions of objects in the tensor products. As $\otimes$ and $\oplus$ are monoidal products we can apply them to morphisms as well as objects which is captured by the constructors $\otimes$C and $\oplus$C.

3 Bipermutative categories

Bipermutative categories are special cases of rig categories. They are *semi-strict*, meaning that the associators and unitors of both monoidal structures have to hold strictly, as well as the annihilators and *one* of the distributors. By the coherence theorem for rig categories [3], every rig category is rig equivalent to a bipermutative one. The language Π defines the syntax for rig categories, and due to the coherence theorem, structural equivalences can all be safely regarded as the identity.

Expressing this property in Agda comes with one major challenge: monoidal categories (such as the structures involved in a rig category) are implemented weakly, meaning that each use of a coherence condition or structural isomorphism must be declared explicitly. Even if their proofs are simple, having to explicitly apply these equivalences inside proofs is "overwhelmingly tedious" [1] and entirely unnecessary in the case of bipermutative categories. To avoid this, we use a feature in Agda called *rewrite rules*.

Agda's rewrite rules. Rewrite rules [2] in Agda are a tool for declaring definitional equalities from any user-defined equivalences which can then be used by the Agda type checker. In a two-step procedure, we first declare a user-defined data type (implementing a relation) to be the target type of a rewrite rule. Second, we declare an instance of this data type (i.e. a relation between two concrete terms) as a definitional equality.

Our plan is to use the type of (reversible) morphisms in rig categories as the target type, and the corresponding structural identities as rewrite rules. Let us consider the structure

isomorphisms for the ⊗ tensor product (as introduced in Equation 1), together with the annihilators coming from the rig category structure:

```
assoc⊗r : ((A ⊗ B) ⊗ C) ↔ (A ⊗ (B ⊗ C))

unit⊗l : (A : PiTy) → (one ⊗ A) ↔ A
unit⊗r : (A : PiTy) → (A ⊗ one) ↔ A
```

(a) Associativity and units for ⊗.

```
annihilateR : (A ⊗ zero) ↔ zero
annihilateL : (zero ⊗ A) ↔ zero
```

(b) Annihilators.

So far, these terms express the weak versions of the isomorphisms. We now declare them strict by first indicating the relation _↔_ as potential target type of a rewrite rule and then adding the above isomorphisms as rewrite rules and thus enforcing them to hold strictly in the framework. Both of these steps are indicated by the {-# REWRITE #-} pragma:

```
{-# BUILTIN REWRITE _↔_ #-}
{-# REWRITE assoc⊗r unit⊗l unit⊗r annihilateR annihilateL #-}
```

This turns equivalences into actual identities, and every time the left hand side of one of the equations occurs in the program it is replaced by right hand side automatically. In the following, we use rewrite rules for all strict isomorphisms in rig categories, including the equivalences we have just shown together with the strict monoidal structure for the other tensor product ⊕. And we go even further.

In addition to rewrite rules for equivalences on *objects*, we now use them to declare identities between *morphisms* in the category, too. In strict monoidal categories we are not only interested in the domain and codomain types of structural equivalences being the same but also the equivalences *themselves* being equal to the identity morphism. Equivalences between morphisms in a category are specified by the following type of 2-morphisms:

```
data _⇔_ : {A B : Ob} → (A ↔ B) → (A ↔ B) → Set
```

(We omit the constructors for 2-morphisms here, for simplicity.)

We use this data type to state 2-isomorphisms between structural equivalences like associators and unitors and the identity morphism. Some examples of these 2-isomorphisms look like this:

```
assoc⊗r=id : {A B C : Ob} → assoc⊗r {A}{B}{C} ⇔ id (A ⊗ B ⊗ C)
unit⊗l=id : {A : Ob} → unit⊗l A ⇔ id A
unit⊗r=id : {A : Ob} → unit⊗r A ⇔ id A
```

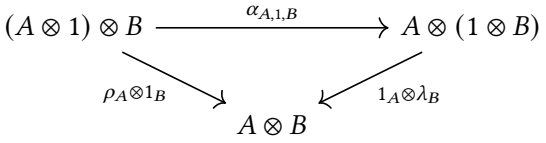
Observe that, to be able to merely *state* these 2-level equivalences, we use the rewrite rules at the object level. As the identity morphism has an equal domain and codomain, any morphism we equate with the identity has to satisfy this property, too. Luckily, the rewrite rules at the object level perform this type coercion automatically, thus the 2-level statements are well typed.

In addition to declaring these equivalences between morphisms, we now declare them as rewrite rules:

```
{-# REWRITE assoc⊗r=id unit⊗l=id unit⊗r=id #-}
```

This means that the proofs of any structural equalities (including some coherence conditions) simplify: whenever a coherence condition holds strictly, it is trivially true in Agda, too.

Example 3.1. The triangle equality (shown on the left) which holds in any monoidal category is the identity natural transformation whenever the category is strict. Using rewrite rules, we can express this property by the following (suitably simple) Agda term:




```
triangle : {A B : Ob} → unit⊗r A ⊗ id B
                      ⇔ assoc⊗r {A}{one}{B}
                        ⨟ (id A ⊗ unit⊗l B)
triangle = id
```

From now onwards we use rewrite rules for all structural identities of rig categories, both at the object and the morphism level. The next step is to add combinators to the language that can capture quantum behaviour.

4 Adding quantum operations

To express quantum operations, $\sqrt{\Pi}$ adds two generators to the language Π, v and w, subject to the following three equations. We also demonstrate how to implement an s gate, using the new generator w:

```
v : two ↔ two
w : one ↔ one
```

(a) Generators.

```
e1 : (w ⨟ w ⨟ w ⨟ w ⨟ w ⨟ w ⨟ w) ⇔ id one
e2 : v ⨟ v ⇔ x
e3 : v ⨟ s ⨟ v ⇔ (w ⨟ w) • (s ⨟ v ⨟ s)
```

(b) Equations.

```
s : two ↔ two
s = id one ⊕C (w ⨟ w)
```

(c) Example S-gate.

Together with the swap operations for the two tensor products, these three equations hold actual computational content for any equality in the language. With the help of rewrite rules, we can now prove properties about the semantics of $\sqrt{\Pi}$ using these computationally relevant equivalences only without the need to explicitly care about structural bookkeeping.

Example 4.1. As an example, we will consider the following equation which proves that we can move a controlled Z gate through the application of an S gate on the control qubit. This equations holds in the system $\sqrt{\Pi}$, and the statement corresponds to equation (A8) in the original paper [1]. The equivalences used in this proof are summarised in Appendix A.

$$
\begin{aligned}
\mathrm{Ctrl}\ Z \circ (\mathrm{S} \otimes \mathrm{Id}) &= \mathrm{SWAP} \circ \mathrm{Ctrl}\ Z \circ (\mathrm{S} \otimes \mathrm{Id}) && \text{(Lemma A.1)} \\
&= \mathrm{SWAP} \circ \mathrm{Ctrl}\ Z \circ (\mathrm{Id} \otimes \mathrm{S}) \circ \mathrm{SWAP} && \text{(Naturality SWAP)} \\
&= \mathrm{SWAP} \circ (\mathrm{Id} \otimes \mathrm{S}) \circ \mathrm{Ctrl}\ Z \circ \mathrm{SWAP} && \text{(Lemma A.2)} \\
&= (\mathrm{S} \otimes \mathrm{Id}) \circ \mathrm{SWAP} \circ \mathrm{Ctrl}\ Z \circ \mathrm{SWAP} && \text{(Naturality SWAP)} \\
&= (\mathrm{S} \otimes \mathrm{Id}) \circ \mathrm{Ctrl}\ Z && \text{(Lemma A.1)}
\end{aligned}
$$

The proof in Agda consists of exactly the same steps as the pen-and-paper proof, and no more. This proof term records the state after every step as well as the property used in

each step (in between the =[_]> brackets). The code snippet also makes extensive use of congruence to indicate to which particular subterm an equality is applied to.

```
A8 : ctrl z ⨟ s ⊗C id two ⇔ (s ⊗C id two) ⨟ ctrl z
A8 = ctrl (p -one) ⨟ s ⊗C id two
  =[ cong (λ x → x ⨟ s ⊗C id two) (sym (A-1 -one)) ]>                  – Lemma A.1
    swap ⨟ ctrl (p -one) ⨟ swap ⨟ (s ⊗C id two)
  =[ cong (λ x → swap ⨟ ctrl (p -one) ⨟ x) (swap⊗C {c1 = s}{id two}) ]> – naturality swap
    swap ⨟ ctrl (p -one) ⨟ (id two ⊗C s) ⨟ swap
  =[ cong (λ x → swap ⨟ x ⨟ swap) (sym (A-2 -one i)) ]>                – Lemma A.2
    swap ⨟ (id two ⊗C s) ⨟ ctrl z ⨟ swap
  =[ cong (λ x → x ⨟ ctrl z ⨟ swap) (swap⊗C {c1 = id two}{s}) ]>       – naturality swap
    (s ⊗C id two) ⨟ swap ⨟ ctrl z ⨟ swap
  =[ cong (λ x → (s ⊗C id two) ⨟ x) (A-1 -one) ]>                      – Lemma A.1
    (s ⊗C id two) ⨟ ctrl z []
```

References

[1] Jacques Carette, Chris Heunen, Robin Kaarsgaard, and Amr Sabry. With a few square roots, quantum computing is as easy as pi. *Proc. ACM Program. Lang.*, 8(POPL), January 2024.

[2] Jesper Cockx. Type theory unchained: Extending agda with user-defined rewrite rules. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 175, 2020.

[3] Donald Yau. *Bimonoidal Categories, E_n-Monoidal Categories, and Algebraic K-Theory: Volume I: Symmetric Bimonoidal Categories and Monoidal Bicategories*, volume 283 of *Mathematical Surveys and Monographs*. American Mathematical Society, 2024.

A Equivalences

Let s and t be scalars and $\mathsf{P}(x) = \mathsf{Id} \oplus s$. Then:

(1) $\mathsf{SWAP} \circ \mathsf{Ctrl}\ \mathsf{P}(s) \circ \mathsf{SWAP} = \mathsf{Ctrl}\ \mathsf{P}(s)$. In Agda:

```
A-1 : (s : one ↔ one) → swap ⨟ ctrl (p s) ⨟ swap ⇔ ctrl (p s)
```

In the original paper, this corresponds to Lemma 10 (v).

(2) $\mathsf{Ctrl}\ \mathsf{P}(s) \circ (\mathsf{Id}_{I \oplus I} \otimes \mathsf{P}(t)) = (\mathsf{Id}_{I \oplus I} \otimes \mathsf{P}(t)) \circ \mathsf{Ctrl}\ \mathsf{P}(s)$. In Agda:

```
A-2 : (s t : one ↔ one) → id two ⊗C (p t) ⨟ ctrl (p s) ⇔ ctrl (p s) ⨟ id two ⊗C (p t)
```

In the original paper, this corresponds to Lemma 10 (vii).

B Definitions

Given a scalar $s : I \to I$ and a morphism $f : X \to Y$, *scalar multiplication* $s \bullet f$ is defined as: $\lambda_\otimes \circ s \otimes f \circ \lambda_\otimes^{-1} : X \to Y$. In Agda, it looks like this:

```
_•_ : one ↔ one → A ↔ B → A ↔ B
_•_ {A}{B} s f = sym (unit⊗l A) ⨟ s ⊗C f ⨟ unit⊗l B
```