Evaluating Linear Functions to Symmetric Monoidal Categories – Version With Additional Details

Jean-Philippe Bernardy
University of Gothenburg
Gothenburg, Sweden
jean-philippe.bernardy@gu.se

Arnaud Spiwack
Tweag
Paris, France
arnaud.spiwack@tweag.io

Abstract

A number of domain specific languages, such as circuits or data-science workflows, are best expressed as diagrams of boxes connected by wires. Unfortunately, functional languages have traditionally been ill-equipped to embed this sort of languages. The Arrow abstraction is an approximation, but we argue that it does not capture the right properties.

A faithful abstraction is Symmetric Monoidal Categories (SMCs), but, so far, it hasn't been convenient to use. We show how the advent of linear typing in Haskell lets us bridge this gap. We provide a library which lets us program in SMCs with linear functions instead of SMC combinators. This considerably lowers the syntactic overhead of the EDSL to be on par with that of monadic DSLs. A remarkable feature of our library is that, contrary to previously known methods for categories, it does not use any metaprogramming.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; • **Theory of computation** → *Models of computation.*

Keywords: Linear Types, Arrows, Symmetric Monoidal Categories, Embedded Domain-Specific Languages

ACM Reference Format:
Jean-Philippe Bernardy and Arnaud Spiwack. 2021. Evaluating Linear Functions to Symmetric Monoidal Categories – Version With Additional Details . In *Unpublished*. ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Preamble to the supplementary material. This version contains additional details. Such material is always marked with a gray background, such as the one which is observable in this note.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

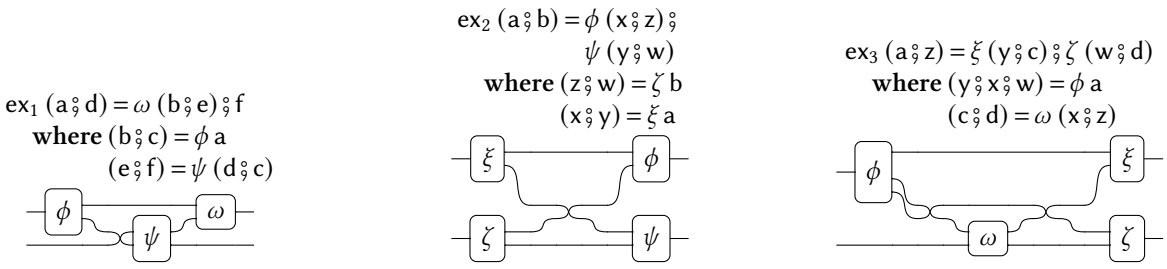
1 Introduction

Parable. Frankie is designing a domain-specific language (DSL), and by working out examples on paper, realises that the best way to describe objects in that DSL is by box-and-wires diagrams, similar to those in Fig. 1. The story does not say what Frankie intends to use the DSL for. Maybe it has to do with linear algebra, parallel computing, or even quantum computations (see Section 4): this kind of pattern occurs in many contexts. Following accepted functional programming methodologies, Frankie searches for the right abstraction and finds out that Symmetric Monoidal Categories (SMC for short) capture said diagrams precisely [19, Section 3]. Accordingly, Frankie starts coding examples using the combinators of SMCs (Fig. 2b), but disappointment is great after writing a few examples: everything is expressed in point-free style, resulting in cryptic expressions such as $(\xi \times \zeta) \circ \bar{\alpha} \circ (\mathsf{id} \times (\alpha \circ (\sigma \times \mathsf{id}) \circ \bar{\alpha} \circ (\mathsf{id} \times \omega) \circ \alpha \circ (\sigma \times \mathsf{id}))) \circ \alpha \circ (\phi \times \mathsf{id})$ for the boxes-and-wires diagram of Fig. 1c. It becomes obvious to Frankie why programming languages have variables: in a language with variables, the same example can be expressed much more directly. Something like:

$$\begin{aligned} \mathsf{ex}_3\ (a \fatsemi z) &= \xi\ (y \fatsemi c) \fatsemi \zeta\ (w \fatsemi d) \\ &\quad \textbf{where}\ (y \fatsemi x \fatsemi w) = \phi\ a \\ &\qquad\quad (c \fatsemi d) = \omega\ (x \fatsemi z) \end{aligned}$$

Now, Frankie could roll-out a special-purpose language for SMCs with variables, together with some compiler, and integrate it into company praxis. But this would be quite costly! For instance, Frankie would have to figure out how to share objects between the DSL and the host programs. Deploying one's own compiler can be a tricky business.

But is it, really, Frankie's only choice? Either drop lambda notation and use point-free style, or use a special-purpose compiler to translate from lambda notation to SMCs? In this paper, we demonstrate that no compromise is necessary: Frankie can use usual functions to encode diagrams. Specifically, we show how to evaluate *linear* functions to SMC expressions. We do so by pure evaluation within Haskell. We require no external tool, no modification to the compiler nor metaprogramming of any kind. This makes our solution particularly lightweight, and applicable to every functional programming language that supports linear types. Even though we specifically target Linear Haskell [3], our



(a) $(\omega \times \mathrm{id}) \circ \bar{\alpha} \circ (\mathrm{id} \times (\psi \circ \sigma)) \circ \alpha \circ (\phi \times \mathrm{id})$ **(b)** $(\phi \times \psi) \circ \bar{\alpha} \circ (\mathrm{id} \times (\alpha \circ (\sigma \times \mathrm{id}) \circ \bar{\alpha})) \circ \alpha \circ (\xi \times \zeta)$ **(c)** $(\xi \times \zeta) \circ \bar{\alpha} \circ (\mathrm{id} \times (\alpha \circ (\sigma \times \mathrm{id}) \circ \bar{\alpha} \circ (\mathrm{id} \times \omega) \circ \alpha \circ (\sigma \times \mathrm{id}))) \circ \alpha \circ (\phi \times \mathrm{id})$

Figure 1. A few SMC morphisms, their encoding as functions, and their string diagram representations.

technique works in any other functional languages with linear types, such as Idris2 [5] or Granule [13].

We make the following contributions:

- We give a linearly typed API to construct SMC morphisms (Section 3). This API is only 5 functions long and allows the programmer to use the name-binding features of Haskell to name intermediate results.
- We demonstrate with concrete applications how our API lets one use Haskell's functions and variables to concisely define SMC morphisms (Section 4).
- We describe an implementation of our API, and prove its correctness (Section 5).
  This implementation was tested on all the examples shown in this paper. In particular, whenever we show a function and a corresponding diagram, as in Fig. 1, our library was used to automatically generate an SMC representation, which was in turn converted to a diagram, and imported to the LaTeX source code of the paper. In this sense, this paper is self-testing.
  The library is available on the Hackage repository: https://hackage.haskell.org/package/linear-smc.

The rest of the paper discusses salient points and related work (Section 6), before concluding in Section 7. Before any of this, we review the underlying concepts and introduce our notations for them (Section 2).

2 Notations and Conventions

In this section we recall the notions of category theory necessary to follow our development and examples. In addition we explain our notation for morphisms and conventions for diagrams.

2.1 Categories

The fundamental structure is that of a category (Fig. 2a). In general a category k is composed of objects and morphisms, but here we take objects to be types satisfying a specific constraint Obj. This choice is convenient because it lets us make the type of Haskell functions an instance of the Category class. A morphism from a to b is a value of type k a b, which we suggestively note $a \overset{k}{\rightsquigarrow} b$. Categories are additionally equipped with an identity at every type (id), which is represented in diagrams as a line. Additionally, categories have morphism composition ($\circ$), represented by connecting morphisms with a line (Fig. 3). This representation neatly captures the laws of categories: morphisms are equivalent iff they are represented by topologically equivalent diagrams. (For instance, composing with the identity simply makes a line longer, and stretching a line is a topology-preserving transformation.) In this paper we follow the usual convention for the directions, even though it means that the layout of diagrams is inverse to that of Haskell expressions. That is, one can think of information as flowing from right-to-left in the expression $f \circ g$, but left-to-right in the diagram representing it.

Even though many applications depend crucially on Obj constraints, they are often lengthy, and orthogonal to our main points. Thus, to minimise clutter, most of the time we omit these Obj constraints. To recover them, one should add an Obj constraint for every relevant type variable, as well as for the unit type. Additionally, for SMCs (introduced in Section 2.2 below), one needs closure under monoidal product.

2.2 Symmetric Monoidal Categories

Our main objects of study are Symmetric Monoidal Categories (abbreviated as SMC throughout the paper). They feature a unit object and the monoidal product (often also called tensor product), written $a \otimes b$. In general the unit can be any type, and the product can be any type family, but it is sufficient for our applications to let the unit object be the unit type (written ()) and the monoidal product as the product type of Haskell (a, b). SMCs provide a number of ways to manipulate the product of objects. First, arbitrary morphisms $f : a \overset{k}{\rightsquigarrow} b$ and $g : c \overset{k}{\rightsquigarrow} d$ can be combined using the ($\times$) combinator: $f \times g : (a \otimes c) \overset{k}{\rightsquigarrow} (b \otimes d)$. This combinator is most often also called a product. In this paper we use different symbols for the product action on morphisms $f \times g$ and on types $a \otimes b$, hopefully minimising confusion. In diagrams, the product of

```
class Category k where
  type Obj k :: Type → Constraint
  id  :: Obj k a ⇒ a ↝k a
  (∘) :: (Obj k a, Obj k b, Obj k c) ⇒
         (b ↝k c) → (a ↝k b) → a ↝k c
```

(a) Category structure

```
class (Category k) ⇒ Monoidal k where
  (×) :: (a ↝k b) → (c ↝k d) → (a ⊗ c) ↝k (b ⊗ d)
  σ   :: (a ⊗ b) ↝k (b ⊗ a)
  α   :: ((a ⊗ b) ⊗ c) ↝k (a ⊗ (b ⊗ c))
  ᾱ   :: (a ⊗ (b ⊗ c)) ↝k ((a ⊗ b) ⊗ c)
  ρ   :: a ↝k (a ⊗ ())
  ρ̄   :: (a ⊗ ()) ↝k a
```

(b) Symmetric Monoidal Category structure

Figure 2. Categorical structures.

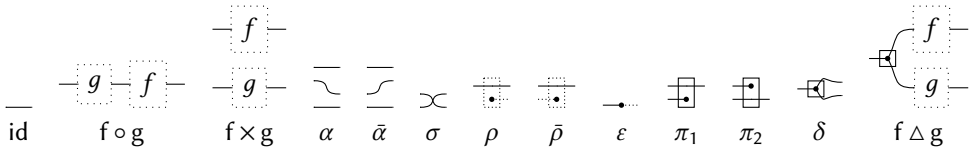


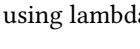
Figure 3. Diagram-Morphism correspondence.

morphisms is represented by laying out the diagram representations of the operands on top of each other. This means that the product morphism has two lines as output and input. In general we allow drawing parallel lines in place of a single line if the corresponding object is a monoidal product. Consequently, the rest of the combinators —associators (α and $\bar{\alpha}$), unitors (ρ and $\bar{\rho}$) and swap (σ)— can be drawn as a (small) descriptive network of lines rather than as abstract boxes. For instance, more tightly associated products are represented by closer parallel lines, and the associators (α and $\bar{\alpha}$) regroup lines accordingly. The purpose of unitors is to introduce or eliminate the unit object, whose carrying lines are drawn dotted. Finally the σ morphism exchanges objects in a product. The reader can refer to Fig. 2b for a summary, and the corresponding diagram representations are shown in Fig. 3. As in the case of simple categories, a great advantage of this diagrammatic notation is that diagrams which can be transformed into one another by continuous deformation (including the removal of disconnected dotted lines) represent equivalent morphisms. This property makes the laws of SMCs intuitive, and because they are extensively documented elsewhere [1], we won't repeat them here. We clarify however that lines can pass each other freely: knots are not taken into account when checking topological equivalence. (For example, two consecutive σ cancel: $\sigma \circ \sigma = \mathsf{id}$.) This property corresponds to the "symmetric" qualifier in "*Symmetric* Monoidal Categories", and it is important to us because it means that one need not worry about the order of binding or use of variables when using lambda notation to describe morphisms.

```
class Monoidal k ⇒ Cartesian k where
  π1  :: (a ⊗ b) ↝k a
  π2  :: (a ⊗ b) ↝k b
  ε   :: a ↝k ()
  δ   :: a ↝k (a ⊗ a)
  (△) :: (a ↝k b) → (a ↝k c) → a ↝k (b ⊗ c)
```

Figure 4. Cartesian structure

2.3 Cartesian Categories

Another key concept is that of cartesian categories (Fig. 4). Even though they are often presented as standalone structures, we instead present them as a layer on top of SMCs. More precisely, we add only new morphisms: no new way to *combine morphisms* is necessary. (In the boxes-and-wires metaphor, we add only new boxes, and no layout rule is added.) A minimal set of such new morphisms is comprised of ε and δ, which respectively discard and duplicate an input. However, it is useful to consider alternative presentations, which can be more convenient, depending on the purpose. Instead of ε, one can use projections (π_1 and π_2), with $\pi_1 = \bar{\rho} \circ (\mathsf{id} \times \varepsilon)$ and likewise for π_2. Likewise, but independently, one may use the combinator ($\triangle$) instead of δ, with $f \triangle g = (f \times g) \circ \delta$. Our diagram notation makes the latter two variants indistinguishable, while the former two are equivalent under pruning of dotted lines:  .

It is enlightening to consider what becomes of the correspondence between diagram (topological) equivalence and morphism (algebraic) equivalence in the presence of the

above laws. For ε, the metaphor can be sustained: continuous deformation of lines involving $\multimap$ capture its laws. For δ, the topological metaphor begins to break down. Morphisms can commute with δ in the following way: $(f \times f) \circ \delta = \delta \circ f$. This breakdown has consequences for computational applications, as we discuss in Section 6.2.

2.4 Linear Types

We rely on linear types in Haskell in an essential way. Indeed, every linear function can be interpreted in terms of an SMC. This is a well known fact, proven for example by Szabo [21, Ch. 3] or Benton [2]. Unfortunately it does not mean that we have nothing to do. Indeed, the above result, as it stands, only means that one can obtain an SMC representation from another *representation* as a (well-typed) lambda term. Such a term is, indeed, constructed by a compiler, but it is in general not made available to the programs themselves: some form of metaprogramming would be required. Unfortunately, outside the Lisp family, such metaprogramming facilities are often brittle or non-existent. For instance, the Template Haskell API is a direct reflection of the internal representation of source code in use by the Glasgow Haskell Compiler, and consequently the user-facing API changes whenever this internal representation changes.

In this paper we use Linear Haskell as host language, and borrow its semantics and notations. We refer to Bernardy et al. [3] if any doubt should remain, but what the reader should know is that linear functions are denoted with a lollipop ($\multimap$), and the pointy-headed arrow ($\to$) corresponds to usual functions, which can use their argument any number of times. A notable feature of Linear Haskell is that unrestricted inputs can be embedded in data types (which can themselves be handled linearly). We make use of this feature in our implementation (Section 5). In sum, any language with the above feature set is sufficient to host our interface and implementation. In particular, we do not make use of the ability of Linear Haskell to quantify over the multiplicity (linear or unrestricted) of function types.

3 Interface

With all the basic components in place, we can now reveal the interface that we provide to construct the morphisms of a symmetric monoidal category k using lambda notation. We introduce a single abstract type: P k r a, where r is a type variable (unique for the morphism under construction) and a is an object of the category k. Values of the type P k r a are called *ports carrying* a. In the boxes-and-wires metaphor, ports are the output wires of boxes. Indeed, the type of morphisms $a \overset{k}{\rightsquigarrow} b$ is encoded as functions of type P k r a $\multimap$ P k r b. However, the type P k r a is abstract: it is manipulated solely *via* the combinators of Fig. 5. (This is enforced according to standard Haskell praxis: the definitions are hidden behind a module boundary, which exports only the prescribed API.)

```
type P  :: (Type → Type → Type) → Type → Type → Type
unit    :: P k r ()
split   :: P k r (a ⊗ b) ⊸ (P k r a, P k r b)
merge   :: (P k r a, P k r b) ⊸ P k r (a ⊗ b)
encode  :: (a ↬ b) → (P k r a ⊸ P k r b)
decode  :: (∀ r. P k r a ⊸ P k r b) → (a ↬ b)
```

Figure 5. The port API

Our bread and butter are the split and merge combinators, which provide the ability to treat ports of type P k r (a ⊗ b) as a pair of ports. In fact, split and merge are ubiquitous enough to deserve a shorthand notation, suggestive of the pair-like character of P k r (a ⊗ b):

- We write (a ⅋ b) for merge (a, b)
- We also use (a ⅋ b) as a pattern, and interpret it as a call to split. For instance, **let** (a ⅋ b) = f **in** u means **let** (a, b) = split f **in** u

Likewise, the presence of unit means that ports of type P k r () can be created from thin air, which is useful to embed constants. Finally and crucially, encode and decode provide means to convert back and forth between morphisms of an SMC ($a \overset{k}{\rightsquigarrow} b$) and (P k r a $\multimap$ P k r b), the corresponding linear functions. We see in the type of decode how the type variable r is introduced, ensuring that ports coming from different functions are not mixed. This interface is guaranteed to satisfy the following properties:

Definition 3.1. Laws of the interface

- split and merge are inverses: split (merge p) = p and merge (split p) = p
- encode and decode are inverses: encode (decode f) = f and decode (encode p) = p
- encode is a functor: encode id = id and encode ($\phi \circ \psi$) = encode $\phi \circ$ encode ψ
- encode is compatible with products: encode ($\phi \times \psi$) (a ⅋ b) = (encode ϕ a ⅋ encode ψ b)
- unit corresponds to unitors: encode ρ a = (a ⅋ unit) and encode $\bar{\rho}$ (a ⅋ unit) = a
- σ, α and $\bar{\alpha}$ are consistent between Haskell and the embedded category:
  - encode σ (a ⅋ b) = (b ⅋ a)
  - encode α ((a ⅋ b) ⅋ c) = (a ⅋ (b ⅋ c))
  - encode $\bar{\alpha}$ (a ⅋ (b ⅋ c)) = ((a ⅋ b) ⅋ c)

Stating the laws which involve products does require a bit of care. For instance, it would not have been type-correct to write encode (f × g) = encode f × encode g nor encode $\sigma = \sigma$: going through split and merge is necessary.

Another aspect to consider is that many of these laws refer to an equality on ports. Because the type of ports is abstract, we cannot define it yet: its concrete definition will be provided together with the concrete definition of ports.

However, we can already give an intuition for it in terms of diagrams: two ports are equal if they are one and the same in the diagram. Even it is abstract, we can already reason with this equality via the following property: two extensionally equal functions on ports will decode to the same morphism. Formally: $(\forall\, x.\, f\, x = g\, x) \to \text{decode}\, f = \text{decode}\, g$.

Without introducing any additional concept, we can already observe some benefits of the above interface. First, one can use all the facilities of a higher-order language to construct elements of $a \overset{k}{\rightsquigarrow} b$, even though k does not have an internal notion of functions (it need not be a closed category). We owe this benefit to the host language evaluation, which takes care of evaluating all intermediate redexes. It can be illustrated by the existence of currying combinators:

```
curry   :: (Monoidal k) ⇒ (P k r (a ⊗ b) ⊸ P k r c)
        ⊸ (P k r a ⊸ P k r b ⊸ P k r c)
curry f a b = f (a ⨾ b)
uncurry :: (Monoidal k) ⇒ (P k r a ⊸ P k r b ⊸ P k r c)
        ⊸ (P k r (a ⊗ b) ⊸ P k r c)
uncurry f p = case split p of (a, b) → f a b
```

Second, if the category k happens to be cartesian, then we can freely copy and discard ports. This is done by encoding ε and δ, as follows:

```
copy :: (Cartesian k) ⇒ P k r a ⊸ P k r (a ⊗ a)
copy = encode δ
discard :: (Cartesian k) ⇒ P k r a ⊸ P k r ()
discard = encode ε
```

It is worth stressing that copy and discard are not part of the abstract interface. Indeed, in the above the morphisms δ and ε are treated as black boxes by our implementation, just like any other morphism of k would be. Consequently the implementation does not assume that any law holds for them, and in particular it cannot commute any morphism with (this instance of) δ using the law $f \times f \circ \delta = \delta \circ f$. We come back to this aspect in Section 6.2. More generally, thanks to the encode combinator, every morphism of k can be turned into a Haskell function on ports.

4 Applications

In this section, we put the port API of Fig. 5 to use. Through two examples of diagrammatic languages, we illustrate how convenient it is to describe box-and-wire diagrams as functions on ports.

4.1 Quantum Circuits

In quantum computing one of the common ways to represent programs is as *quantum circuits*. Take, for instance, the circuit of Fig. 6, which is an implementation of the Toffoli gate in terms of simpler quantum gates.

For our purposes, it suffices to treat the atomic gates in Fig. 6 as abstract. Regardless, if a reader may be interested in looking up their definitions, the gate H stands for the Hadamard gate, T for the T gate, and $\oplus$ for the controlled-not gate.[1] Quantum circuits closely resemble traditional Boolean circuits except that a circuit represents not a Boolean function, but a unitary matrix on some finite dimensional $\mathbb{C}$-vector space. For our purposes, unitary matrices have two important properties. First, they form an SMC, which we call U. (This is why quantum circuits can be written as boxes-and-wires diagrams.)

A possible implementation of the U category is to let $a \overset{U}{\rightsquigarrow} b$ be a matrix whose indices range a and b.

```
data U a b = U {fromM :: Array (a, b) ℂ}
```

Thus this means in particular that all objects in this category must be finite types: Finite a = (Bounded a, Ix a, Eq a). This way we can construct matrices using the following function:

```
tabulate :: (Finite a, Finite b) ⇒ (a → b → ℂ) → a ⇝ b
tabulate f = U (array ((minBound, minBound),
                       (maxBound, maxBound))
    [((i, j), f i j) | i ← inhabitants, j ← inhabitants])
```

Besides, the main tool for implementation is the Kronecker delta:

```
delta :: (Eq a) ⇒ a → a → ℂ
delta x y = if x ≡ y then 1 else 0
```

We can then construct the Monoidal U instance:

```
instance Category U where
  type Obj U = Finite
  id = tabulate delta
  U g ∘ U f = tabulate (λ i j → summation
                          (λ k → f ! (i, k) * g ! (k, j)))
instance Monoidal U where
  U f × U g = tabulate (λ (a, c) (b, d) → f ! (a, b) * g ! (c, d))
  ρ = tabulate (λ x (y, ()) → delta x y)
  ρ̄ = tabulate (λ (y, ()) x → delta x y)
  α = tabulate (λ ((x, y), z) (x′, (y′, z′)) →
                  delta ((x, y), z) ((x′, y′), z′))
  ᾱ = tabulate (λ (x′, (y′, z′)) ((x, y), z) →
                  delta ((x, y), z) ((x′, y′), z′))
  σ = tabulate $ λ (x, y) (y′, x′) → delta (x, y) (x′, y′)
```

Morphism composition is matrix product, and the product ($\times$) is implemented as the Kronecker product.

To be complete, we also would need to show that each method implemented above preserves the unitary character of matrices. These proofs can be easily looked up, but for the reader who might prefer to reconstruct them, the key property is that a matrix is unitary iff its determinant is 1: norm (det u) = 1. Then one needs to check that this property is preserved by each operation. The properties to invoke are det (u ∘ v) = det u · det v and det (u × v) = (det u) ^ n · (det v) ^ m, where n and m are the respective dimensions of u and v.

[1] Refer for example to https://en.wikipedia.org/wiki/Quantum_logic_gate and https://en.wikipedia.org/wiki/Toffoli_gate.

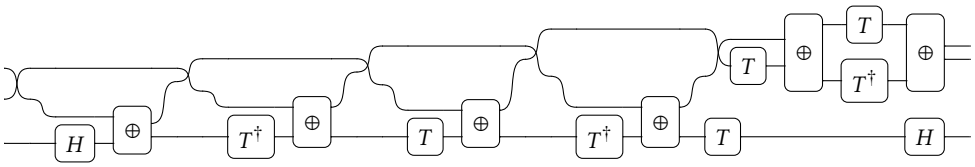


Figure 6. Toffoli gate in terms of H, T and $\oplus$.

Second, unitary matrices can be inverted by taking their conjugate transpose. Notice for example the use of the gate $T^\dagger$ in Fig. 6. It is the conjugate transpose of T. That is, $T^\dagger$ is not a primitive gate, but one defined in terms of T using the function

```
conjugateTranspose :: U b a ⟶ U a b
```

It would be inconvenient to have to return to the low-level SMC interface every time we want to invert a matrix: what we really want is to lift the U-level interface to ports (P U) once and for all, then work entirely with ports. Fortunately, we can do just that. The only difference with lifting simple morphisms ($a \overset{U}{\leadsto} b$) is that lifting conjugateTranspose yields a higher-order function:

```
invert :: (∀ s. P U s a ⊸ P U s b) ⟶ (∀ r. P U r b ⊸ P U r a)
invert f = encode (conjugateTranspose (decode f))
```

Consequently we do not have to encode the diagram of Fig. 6 using the methods of the Monoidal class, but we can use the more familiar lambda notation, manipulating ports. We do so assuming the gates H, T, and $\oplus$, which we can leave abstract with the following types:

```
H :: P U r Bool ⊸ P U r Bool
T :: P U r Bool ⊸ P U r Bool
(⊕) :: P U r Bool ⊸ P U r Bool ⊸ (P U r Bool, P U r Bool)
```

Now, we can define the Toffoli gate circuit as follows

```
toffoli ::   P U r ((Bool ⊗ Bool) ⊗ Bool)
         ⊸ P U r ((Bool ⊗ Bool) ⊗ Bool)
toffoli c1 c2 x = c1 ⊕ H  x  & λ (c1, x) ⟶
                  c2 ⊕ T† x  & λ (c2, x) ⟶
                  c1 ⊕ T  x  & λ (c1, x) ⟶
                  c2 ⊕ T† x  & λ (c2, x) ⟶
                  c2 ⊕ T  c1 & λ (c2, y) ⟶
                  (T c2 ⊕ T† y) ⨟ (H (T x))
                  where T† = invert T
```

We use explicit β-redexes instead of let-bindings here because we want to reuse some variable names: since using a linear variable makes it unavailable in the remainder of the function, we may freely reuse its name. Unfortunately, Haskell only has recursive lets, so if we were to write **let** (c1 ⨟ x) = c1 ⊕ H x **in** ..., Haskell would try to define both c1 and x recursively, which is not the intended behaviour. To this effect, we use the reverse-order linear application operator (&) which is defined as

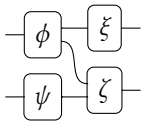


Figure 7. A workflow corresponding to the morphism $(\xi \times \zeta) \circ \alpha \circ (\phi \times \psi)$.

```
(&) :: a ⊸ (a ⊸ b) ⊸ b
x & f = f x
```

This is a specificity of Haskell. In a language with non-recursive lets the definition of toffoli would look even more natural.

4.2 Workflow Orchestration

Consider a type Step a b representing computations from type a to type b: a value of type Step a b may be some Haskell function, or it can run an external command. Whatever it is, we make the assumption that the side effects embedded in a Step are commutative. That is, it never matters if step ϕ is run before step ψ or the other way around. And, in fact, if there is no data dependencies between ϕ ant ψ, we want to run them in parallel.

What we want to do, in this scenario, is to compose individual steps together to form bigger computations, typically called a *workflow*. In Fig. 7 we show a simple, albeit typical, workflow.

What would a DSL to that effect look like? A first attempt may be to organise the DSL around a monad M, and define the workflow of Fig. 7 as follows:

```
type Workflow a b = a ⟶ M b
workflowM :: Workflow (A, B) (C, D)
workflowM (a, b) = do
  (x, y) ← ϕ a
  z ← ψ b
  c ← ξ x
  d ← ζ (y, z)
  return (c, d)
```

The problem with this monadic DSL, however, is that it forces us to fully sequentialise our workflow: ϕ runs before ψ, which runs before ξ, which runs before ζ. This is wasteful: a glance at Fig. 7 makes it obvious that ϕ and ψ can be run in parallel, as well as ξ and ζ, etc. Running independent steps in parallel may be crucial to performance. But the monad abstraction

makes the inherent parallelism fundamentally unrecoverable.

To improve upon this state of affairs, one could attempt to leverage an applicative functor structure that M may exhibit. Accordingly one can recover parallelism as follows:

```
workflowA :: Workflow (A, B) (C, D)
workflowA (a, b) = do
  ((x, y), z) ← (,) <$> ϕ a <*> ψ b
  (c, d) ← (,) <$> ξ x <*> ζ (y, z)
  return (c, d)
```

This is the style advocated, in the context of database query batching, by the Haxl library [11]. GHC even features an extension (ApplicativeDo [12]) that automatically translates code written using the **do**-notation (as in workflowM) to use applicative combinators for parallel commands as workflowA does. Unfortunately, even workflowA doesn't fully expose all the parallelism opportunities: workflowA will run both ϕ and ψ in parallel, but it will wait until both are completed before starting either ξ or ζ. But only the result of ϕ is necessary to run ξ. If ψ takes more time to run than ϕ, then this is wasteful.

One could try to rewrite the workflow as follows:

```
workflowA′ :: Workflow (A, B) (C, D)
workflowA′ (a, b) = do
  ((y, c), z) ← (,) <$> part1 <*> ψ b
  d ← ζ (y, z)
  return (c, d)
  where part1 = do (x, y) ← ϕ a
                   c ← ξ x
                   return (y, c)
```

Now ξ can start as soon as ϕ completes, and run in parallel with ψ. But ζ has to wait for ξ to complete before it can start. In sum, the combined Applicative-Monadic interface prevents *any* implementation to fully expose the parallelism opportunities inherent in the workflow.

Haskell offers another abstraction, called *arrows* [9], to model parallelism. This is how Parès et al. [14] model workflows. In this style, our example would look like:

```
data Workflow a b
instance Arrow Workflow
workflowArr :: Workflow (A, B) (C, D)
workflowArr = (ϕ *** ψ) >>>
              arr (λ ((x, y), z) → (x, (y, z))) >>>
              (ξ *** ζ)
```

Or, using the built-in notation for arrows [15]

```
workflowArr′ :: Workflow (A, B) (C, D)
workflowArr′ = proc (a, b) → do
  (x, y) ← ϕ ↢ a
  z ← ψ ↢ b
  c ← ξ ↢ x
  d ← ζ ↢ (y, z)
  returnA ↢ (c, d)
```

In workflowArr, like workflowA, ξ must run after ψ. It is also possible to write a version of the workflow which, like workflowA′, has ξ running in parallel with ψ, but ζ must run after ξ. In sum, This arrow-based DSL suffers from the same problem as the applicative DSL: some over-sequentialisation is unavoidable.[2] Indeed, a situation just as this one, in an industrial workflow, was one of the motivations behind this paper. It was impossible to optimise resources usage in that workflow due to the limitation of the arrow abstraction, wasting resources.

In contrast, if workflows are given an SMC instance, all the parallelism of Fig. 7 is exposed and can be exploited by the workflow scheduler.

```
data Workflow a b
instance Monoidal Workflow
workflowSMC :: P Workflow r (K A) ⊸ P Workflow r (K B)
                 ⊸ P Workflow (K C ⊗ K D)
workflowSMC a b = ϕ a  &λ (x ⊗ y) →
                  ψ b  &λ z →
                  ξ x  &λ c →
                  ζ y z &λ d →
                  (c ⊗ d)
```

This version is syntactically close to the monadic workflowM implementation: the chief difference is the use of reverse application (&) instead of the monadic bind. Yet, all the parallelism is retained!

A noteworthy element of this workflow DSL is the presence of K wrappers around the types A, B, C, and D. The rationale is that synchronisation points will be at the level of atomic types, and K indicates such atomic types. That is, if two sub-workflows are connected by the type K (a ⊗ b), then there can be no parallelisation between them. However, if they are connected by K a ⊗ K b, then parallelisation can be discovered by the scheduler. (Another option to identify atomic types would have to let ⊗ be different from the native Haskell product.)

Composable workflows are implemented as IO actions connecting two synchronisation Points. This means in particular that they embed synchronisation primitives (which reside in IO () in Concurrent Haskell):

```
data Workflow a b
  = W {taskRun :: Point a → Point b → IO ()}
```

These Points must be at the level of base types (not products thereof) so that synchronisation is as fine-grained as necessary. Hence, the Point type must be defined by structural induction over types, such that the synchronisation point

[2] This problem with Arrow can be attributed to the arr combinator. Because arr embeds a Haskell function, it is opaque and thus prevents any efficient scheduling strategy between the morphisms connected to it. Of course, for some specific Arrow instances, one can provide the combinators of an SMC and recover a better behaviour when using them instead of arr. However this does not apply when using the arrow notation, because it always desugars to calls to arr.

of a product is the product of synchronisation points. For atomic types, synchronisation can be implemented by any suitable mechanism provided by Haskell. Here we have chosen the MVars of concurrent Haskell [17]. Such an induction can be implemented in Haskell by exploiting the Obj constraint over types. We let it be a type-class HasPoint, with separate instances for products and for base types. The form of base type is required to be K a with **data** K a = K a. If the type Point a is an associated data type of the class HasPoint, we get an inductive definition as desired:

```
class HasPoint a where
  data Point a :: Type
  mkPoint :: IO (Point a)
  connect :: Point a → Point a → IO ()
instance (HasPoint a, HasPoint b) ⇒ HasPoint (a ⊗ b) where
  data Point (a ⊗ b) = (Point a) :* (Point b)
  mkPoint = do
    a ← mkPoint
    b ← mkPoint
    return (a :* b)
  connect (a :* b) (a' :* b') = do
    connect a a'
    connect b b'
instance HasPoint (K a) where
  data Point (K a) = Atom (MVar a)
  mkPoint = Atom <$> newEmptyMVar
  connect (Atom a) (Atom b)
        = forkIO (takeMVar a >>= putMVar b) >> return ()
```

The product (f × g) is implemented by running f and g in separate threads (forking one extra thread). As described above, the composition (f ∘ g) runs f and g in parallel, with a new synchronisation point in-between. This means that if f and g are run as subtasks with fine-grained dependencies: no unnecessary synchronisation happens. The σ morphism is implemented by forwarding data as appropriate. The other ones (α , ρ) follow the same pattern and are omitted for concision.

```
instance Category Workflow where
  type Obj Workflow = HasPoint
  id = W connect
  W f ∘ W g = W $ λ a c → do
    b ← mkPoint
    forkIO (g a b) >> f b c
instance Monoidal Workflow where
  W f × W g = W $ λ (a :* b) (c :* d) → do
    _ ← forkIO (f a c)
    g b d
  σ = W $ λ ((a :* b)) ((c :* d)) → do
    connect a d
    connect b c
  ᾱ = W $ λ ((a :* ((b :* c)))) ((((d :* e)) :* f)) → do
    connect a d
    connect b e
    connect c f
  α = W $ λ (((a :* b) :* c)) ((d :* (e :* f))) → do
    connect a d
    connect b e
    connect c f
  ρ = W $ λ a ((a' :* _)) → connect a a'
  ρ̄ = W $ λ ((a :* _)) a' → connect a a'
```

The above implementation is only a prototype for illustrative purposes. For instance, it causes a synchronisation point to happen even between every two connected atomic tasks. This excessive synchronisation can induce significant overheads in some situations. If this is a concern, one can perform an analysis of the computation graph (say, by first reifying it as a data type) and eliminate unneeded synchronisation points. Additionally, applications may need some mechanism to deal with errors or dead tasks.

A more fundamental limitation of the prototype resides in synchronisation being of the simplest kind: connection between ports is realised by simply forwarding data— always in the same direction. Thus, another extension to the above prototype would be to support more complex protocols (corresponding to other base types than K a). For example, sequential data can be streamed, one element at a time. Query-reply protocols are also a possibility. In this light, we can now examine the question of whether tasks form a cartesian category (in addition to symmetric monoidal). Because the δ morphism corresponds to multiplexing, Workflows can be enthused with a cartesian structure only if all base protocols are multiplexable in their input. (The condition that the unit type is the unit for multiplexing would normally be satisfied as well.)

5 Implementation

In this section we reveal the implementation of our abstract type for the API from Section 3. Unfortunately it is not just a matter of writing down the specification and calculating an implementation: some amount of creativity is required. *The*

```
data FreeCartesian k a b where
  I      :: FreeCartesian k   a a
  (:∘:)  :: FreeCartesian k b c → FreeCartesian k a b
         → FreeCartesian k a c
  Embed  :: k a b → FreeCartesian k a b
  (:△:)  :: FreeCartesian k a b → FreeCartesian k a c
         → FreeCartesian k a (b ⊗ c)
  P₁     :: FreeCartesian k (a ⊗ b) a
  P₂     :: FreeCartesian k (a ⊗ b) b
instance (Monoidal k) ⇒ Monoidal (FreeCartesian k)
instance (Monoidal k) ⇒ Cartesian (FreeCartesian k)
```

Figure 8. Definition of the free cartesian category over an underlying category k, whose morphisms it Embeds. P_1 and P_2 implement respectively π_1 and π_2, while (:△:) implements (△).

*key idea is to represent ports as morphisms from the source (*r*) to the object of interest.* In terms of diagrams, they represent the portion of the diagram which connect the source (on the left) to the port.

Such morphisms may therefore discard part of the input. This means that they are not morphisms of the SMC k, but rather morphisms of the free cartesian category over k (FreeCartesian k r a, see Fig. 8):

```
data P k r a where
  P :: FreeCartesian k r a → P k r a
fromP :: P k r a → FreeCartesian k r a
fromP (P f) = f
```

This free category is implemented as a data type with a constructor for each method in the Cartesian class, plus a constructor to Embed generators from k. A subtlety is that, even though P k r a is used linearly everywhere in the interface, the free cartesian representation that it embeds can be duplicated at will. In Linear Haskell this is subtly noted by using using the → arrow instead of ⊸ in the declaration of P constructor. Consequently, when doing *encode* ϕ, the morphism ϕ must be available *unrestricted*, not just linearly. This is not a problem in practice: even if data cannot be duplicated, closed functions which manipulate such data can be.

With these technical bits out of the way, let us return to the main representational idea: a port for the object *a* is a free cartesian morphism from r to a. Accordingly, the equality on ports is the usual equality of free cartesian categories, but quotiented by equations arising from Embed being an SMC homomorphism:

```
Embed id       = id
Embed (φ ∘ ψ)  = Embed φ ∘ Embed ψ
Embed (φ × ψ)  = Embed φ × Embed ψ
```

etc.

Because P k r a is a morphism from r to a, the encoding from $a \overset{k}{\rightsquigarrow} b$ to P k r a ⊸ P k r b can be thought of as a transformation to continuation-passing-style (CPS), albeit reversed—perhaps a "prefix-passing-style" transformation. For non-linear functions, the encoding would be given by the Yoneda lemma [4] composed with embedding in the free cartesian category. The implementation of the combinators of the interface can then follow the usual (cartesian) categorical semantics of product and unit types:

```
encode φ (P f)   = P (Embed φ ∘ f)
unit             = P ε
split (P f)      = (P (π₁ ∘ f), P (π₂ ∘ f))
merge (P f, P g) = P (f △ g)
```

The most challenging part of the implementation is decode, which converts *linear* functions between ports to morphisms in k.

```
decode f = evalM (reduce (extract f))
extract  :: (∀ r. P k r a ⊸ P k r b) → FreeCartesian k a b
extract f = fromP (f (P id))
```

As usual in CPS, the first step is to complete the computation by passing the identity morphism (extract). Then the obtained FreeCartesian k morphism is projected to the SMC k, which it carries. The next step is reduce, which projects the free cartesian representation to a free SMC representation, referred hereafter as FreeSMC. This is the most difficult operation, and we return to it shortly. The evalM part maps a morphism of FreeSMC k back to a morphism in k —it is the natural inductive definition on the structure of free-SMC morphisms.

```
data FreeSMC k a b where
  I      :: FreeSMC k   a a
  Embed  :: k a b → FreeSMC k a b
  A      :: FreeSMC k ((a ⊗ b) ⊗ c) (a ⊗ (b ⊗ c))
  A'     :: FreeSMC k (a ⊗ (b ⊗ c)) ((a ⊗ b) ⊗ c)
  S      :: FreeSMC k (a ⊗ b) (b ⊗ a)
  U      :: FreeSMC k a (a ⊗ ())
  U'     :: FreeSMC k (a ⊗ ()) a
  (:∘:)  :: FreeSMC k b c → FreeSMC k a b
         → FreeSMC k a c
  (:×:)  :: FreeSMC k a b → FreeSMC k c d
         → FreeSMC k (a ⊗ c) (b ⊗ d)
```

The equality for FreeSMC is quotiented by the same laws regarding Embed as the FreeCartesian representation.

```
evalM :: (Monoidal k) ⇒ FreeSMC k a b → a ↷ᵏ b
evalM I          = id
evalM (f :×: g)  = evalM f × evalM g
evalM (f :∘: g)  = evalM f ∘ evalM g
evalM A          = α
evalM A'         = ᾱ
evalM S          = σ
evalM U          = ρ
evalM U'         = ρ̄
evalM (Embed φ)  = φ
```

5.1 Proving the Implementation Correct

Even though we have not fully described the implementation yet, we know enough to prove it correct. (Indeed, the only remaining uncertainty is in the implementation of reduce, but we already have specified that it must not change the meaning of morphisms, only project them from free cartesian to free SMC representations.)

To begin, we show that decode respects the equality on ports. Indeed, due to this equality being quotiented by Embed being an SMC-homomorphism, a bit of reasoning is necessary to prove that functions over ports which are extensionally equal (with the above equality for outputs) are decoded to equal morphisms:

Lemma 5.1. $(\forall\, x.\, f\, x = g\, x) \to \mathsf{decode}\, f = \mathsf{decode}\, g$

Proof. The idea is that decode subjects all FreeCartesian morphisms to evalM. Because evalM maps representations that are equal under the Embed homomorphism equations to equal morphisms in k, we have our result.

Formally, the implication is proven by a transitive application of number of congruences:

$\forall\, x.\, f\, x = g\, x$

$\to$

$f\,(\mathsf{P}\,\mathsf{id}) = g\,(\mathsf{P}\,\mathsf{id})$

$\to$ *by congruence*

$\mathsf{reduce}\,(f\,(\mathsf{P}\,\mathsf{id})) = \mathsf{reduce}\,(g\,(\mathsf{P}\,\mathsf{id}))$

$\to$*by Lemma 5.2*

$\mathsf{evalM}\,(\mathsf{reduce}\,(f\,(\mathsf{P}\,\mathsf{id}))) = \mathsf{evalM}\,(\mathsf{reduce}\,(g\,(\mathsf{P}\,\mathsf{id})))$

$\to$ *by def.*

$\mathsf{evalM}\,(\mathsf{reduce}\,(\mathsf{extract}\,f)) = \mathsf{evalM}\,(\mathsf{reduce}\,(\mathsf{extract}\,g))$

$\to$ *by def.*

$\mathsf{decode}\,f = \mathsf{decode}\,g$

□

The critical step, which is taken care of by the following lemma, is necessary because we go from an equality on a type where equality is quotiented, to a type where equality is not quotiented.

Lemma 5.2. *if* $x = y$ *then* $\mathsf{evalM}\,x = \mathsf{evalM}\,y$

Proof. We need to show that the terms which we deem equal by quotienting the equality of FreeCartesian are mapped to equal terms by evalM. This is done case by case, and a simple matter of expanding definitions. We show two cases here: the others follow the same patterns.

- $\mathsf{evalM}\,\mathsf{id} = \mathsf{id} = \mathsf{evalM}\,(\mathsf{Embed}\,\mathsf{Id})$
- $\mathsf{evalM}\,(\mathsf{Embed}\,\phi \times \mathsf{Embed}\,\psi) = \mathsf{evalM}\,(\mathsf{Embed}\,\phi) \times \mathsf{evalM}\,(\mathsf{Embed}\,\psi) = \phi \times \psi = \mathsf{evalM}\,(\mathsf{Embed}\,(\phi \times \psi))$

□

We can then prove all the laws given in Definition 3.1.

Theorem 5.3. *The implementation respects the laws stated in Definition 3.1.*

Proof. Each case can be proven by equational reasoning. (In these reduction steps we assume that the P r type forms a cartesian category, obtained by lifting the same structure from FreeCartesian. This simplification means that we can skip many uninformative conversions between the two types using P and fromP.)

- split/merge

  $\mathsf{split}\,(\mathsf{merge}\,(x, y))$
  $=$ *by def*
  $\mathsf{split}\,(x \triangle y)$
  $=$ *by def*
  $(\pi_1 \circ (x \triangle y), \pi_2 \circ (x \triangle y))$
  $=$ *by cartesian category properties*
  (x, y)
- merge/split

  $\mathsf{merge}\,(\mathsf{split}\,f)$
  $=$ *by def*
  $(\mathbf{let}\,(x, y) = ((\pi_1 \circ f), (\pi_2 \circ f))\,\mathbf{in}\,(x \triangle y))$
  $=$ *by evaluation*
  $((\pi_1 \circ f) \triangle (\pi_2 \circ f))$
  $=$ *by cartestian laws*
  f
- decode/encode

decode (encode f)
= *by def*
decode (λ (P x) $\longrightarrow$ P (Embed f $\circ$ x))
= *by def*
evalM (reduce (extract (λ (P x) $\longrightarrow$
P (Embed f $\circ$ x))))
= *by def*
evalM (reduce ((λ x $\longrightarrow$ Embed f $\circ$ x) id))
= *by β-reduction*
evalM (reduce (Embed f $\circ$ id))
= *by property of host language composition*
evalM (reduce (Embed f))
= *by evalM $\circ$ reduce $\circ$ Embed = id*
f

- encode/decode
encode (decode f) (P a)
= *by def of encode*
P (Embed (decode f) $\circ$ a)
= *by def of decode*
P (Embed (evalM (reduce (fromP (f id)))) $\circ$ a)
= *by Embed $\circ$ evalM $\circ$ reduce = id*
f id $\circ$ P a
= *by Covariant Yoneda Lemma (naturality of f)*
f (P a)
The step one way to see that Embed $\circ$ evalM $\circ$ reduce = id is to notice that reduce does not change the meaning of morphisms, only their representation, from free cartesian to free SMC. The composition Embed $\circ$ evalM does the opposite conversion. We have equality because free SMC terms are quotiented by Embed being an homomorphism.
- encode/$\circ$
encode ($\phi \circ \psi$) (P f)
= *by def*
P (Embed ($\phi \circ \psi$) $\circ$ f)
= *by Embed property*
P (Embed $\phi \circ$ Embed $\psi \circ$ f)
= *by def of encode*
encode ϕ (P (Embed $\psi \circ$ f))
= *by def of encode*
encode ϕ (encode ψ (P f))
= *by def of $\circ$*
(encode $\phi \circ$ encode ψ) (P f)
- encode/id
encode id (P f)
= *by definition of encode*
P (Embed id $\circ$ f)
= *by Embed property*
id $\circ$ P f
= *by def*
P f
= *by def*
id (P f)
- encode/merge
encode ($\phi \times \psi$) (P a ⨟ P b)
= *by def*
P (Embed ($\phi \times \psi$) $\circ$ (a $\triangle$ b))
= *by assumption on Embed*
P ((Embed $\phi \times$ Embed ψ) $\circ$ (a $\triangle$ b))
= *by properties of free cartesian categories*
P ((Embed $\phi \circ$ a) $\triangle$ (Embed $\psi \circ$ b))
= *by def*
(encode ϕ (P a) ⨟ encode ψ (P b))
- encode/ρ
encode ρ (P a)
= *by definition of encode*
P (Embed $\rho \circ$ a)
= *by definition of unitor for cartesian categories*
P ((id $\triangle$ ε) $\circ$ a)
= *by property of $\triangle$*
P (a $\triangle$ ($\varepsilon \circ$ a))
= *by property of ε*
P (a $\triangle$ ε)
= *by definitoin of merge*
(P a ⨟ unit)
- encode/ρ'
encode $\bar{\rho}$ (P a ⨟ unit)
= *by def*
P (Embed $\bar{\rho} \circ$ (a $\triangle$ ε))
= *by definition of unitor for cartesian categories*
P ($\pi_1 \circ$ (a $\triangle$ ε))
= *by properties of cartesian categories*
P a
- encode/σ
encode σ (P a ⨟ P b)
= *by def*
P (Embed $\sigma \circ$ (a $\triangle$ b))
= *by assumption on Embed*
P ($\sigma \circ$ (a $\triangle$ b))
= *by properties of free cartesian categories*
P (b $\triangle$ a)
= *by def*
(P b ⨟ P a)
- encode/α
encode α ((P a ⨟ P b) ⨟ P c)
= *by def*
P (Embed $\alpha \circ$ ((a $\triangle$ b) $\triangle$ c))
= *by assumption on Embed*
P ($\alpha \circ$ ((a $\triangle$ b) $\triangle$ c))
= *by properties of free cartesian categories*
P (a $\triangle$ (b $\triangle$ c))
= *by def*
(P a ⨟ (P b ⨟ P c))
- encode/α'

```
  encode ᾱ (P a ⅋ (P b ⅋ P c))
=   by def
  P (Embed ᾱ ∘ (a △ (b △ c)))
=   by assumption on Embed
  P (ᾱ ∘ (a △ (b △ c)))
=   by properties of free cartesian categories
  P ((a △ b) △ c)
=   by def
  ((P a ⅋ P b) ⅋ P c)
```

□

5.2 Characterisation of the Domain of reduce.

As mentioned previously, the bulk of the work is to define (and prove correct) the reduce function, which converts a FreeCartesian representation into a FreeSMC. This reduce function is partial: if its input is not suitable (say if an input is discarded) then there is no SMC representation. Fortunately, we only need to deal with representations which have been constructed using the port interface, namely linear functions built with encode, merge, split and unit. Our plan is then to 1. prove that the extracted morphisms are indeed reducible to the SMC interface, and 2. show how to carry it out algorithmically. We start by addressing the first problem, and this will put us firmly on track to address the second one.

Definition 5.4. A representation $f : \mathsf{FreeCartesian}\ k\ a\ b$ is called linear if it is it defined using only the SMC subset of the cartesian structure.

Definition 5.5. A representation $f : \mathsf{FreeCartesian}\ k\ a\ b$ is called *protolinear* iff it is equivalent, according to the laws of a cartesian category, to a linear representation h.

Theorem 5.6. *For every function* $h : \forall r.\ P\ k\ r\ a \multimap P\ k\ r\ b$*, the morphism* extract h *is a protolinear representation.*

Proof. The idea of the proof is to do an induction on the structure of h. But in general a computational prefix f of h has several outputs. That is, the type of f has the form $P\ k\ r\ a \multimap \bigotimes_i (P\ k\ r\ t_i)$, where where the big circled product operator is a multary version of the monoidal product with right associativity. The components of such products represent ports which are available after the prefix f is run (but h is not complete). Thus, to obtain a protolinear function from f, its outputs must be merged, by a generalised fork (△) function, written $\bigwedge$, and defined as follows:

$$\bigwedge :: \bigotimes_i (P\ k\ a\ t_i) \to P\ k\ a\ (\bigotimes_i t_i)$$
$$\bigwedge (f_1, (\ldots, f_n)) = (f_1 \triangle (\ldots \triangle f_n))$$

When there is a single output port, $\bigwedge$ is the identity, and thus this theorem is a corollary of the generalised form, Lemma 5.7, for a product with one element. □

Lemma 5.7. *If* $f : \forall r.\ P\ k\ r\ a \multimap \bigotimes_i (P\ k\ r\ t_i)$*, then* $\bigwedge (f\ \mathsf{id})$ *is a protolinear representation.*

Proof. First, we need to choose a convenient representation of the function f itself. A first idea could be to use the term representation of Haskell. This would however make for a tedious proof, and to fit our theme, we use a categorical representation for Haskell functions as well. For this purpose, we make the simplifying assumption that functions of the type $\forall r.\ P\ k\ r\ a \multimap P\ k\ r\ b$ can be themselves represented as morphisms in another free SMC, the category of linear functions of Haskell.[3] Additionally, because the type $P\ k\ a\ b$ is abstract, we know that the only possible generators for this SMC are the primitives unit, split, merge and encode: we can assume that other constructions are reduced away by the Haskell evaluator.

Furthermore, this representation can be assumed without loss of generality to take the form of a composition $s_1 \circ \cdots \circ s_n$. (This corresponds to cutting the corresponding diagram in vertical slices s_i, each containing a single generator. By topology-preserving transformations, it is always possible to move generators so that they fall in separate slices.)

In fact, without loss of generality, we assume that each slice s has either of the following forms: 1. $\mathsf{encode}\ \phi \times \mathsf{id}$ 2. $\alpha \circ (\mathsf{split} \times \mathsf{id})$ 3. $(\mathsf{merge} \times \mathsf{id}) \circ \bar{\alpha}$ 4. $\lambda x \to (\mathsf{unit}, x)$. That is, we assume that the generators act on the first component of the slice. (The split and merge cases are composed with associators to preserve the property that the multary monoidal products on the input and output are right-associated.) We can make this assumption because we treat permutations over the monoidal product as separate slices (Of a separate form, referred to as 5. below). Such a slice does not contain any generator; rather its role is to stage the next variable(s) to be acted upon by the next generator.

We can now proceed with the induction. The base case reduces to protolinearity of id, which is obvious. For the induction case, we assume that $\bigwedge (f\ \mathsf{id})$ is protolinear, and show that so is $\bigwedge ((s \circ f)\ \mathsf{id})$, for every function f of type $\forall r.\ P\ k\ r\ a \multimap \bigotimes_i (P\ k\ r\ t_i)$, and every possible slice s.

Let us calculate a reduced form for $(s \circ f)\ \mathsf{id}$ for each case:

- Let $g = \mathsf{encode}\ \phi \times \mathsf{id}$.

[3] To be fair, this property would only be true of an idealised language with linear types (Section 2.4). For an actual programming language, exceptions, non-termination, etc. should be taken into account. In practice, if the function of type $\forall r.\ P\ k\ r\ a \multimap P\ k\ r\ b$ diverges, the reduce function also diverges. This means that we are limited to finite quantum circuits or workflows.

$\bigtriangleup\ (((\text{encode}\,\phi \times \text{id}) \circ \text{f})\ \text{id})$
$=$ *by def of* $\circ$
$\bigtriangleup\ ((\text{encode}\,\phi \times \text{id})\ (\text{f id}))$
$=$ *by expansion of pairs*
$\bigtriangleup\ ((\text{encode}\,\phi \times \text{id})\ (\pi_1\ (\text{f id}), \pi_2\ (\text{f id})))$
$=$ *by definition of* $\times$
$\bigtriangleup\ (\text{encode}\,\phi\ (\pi_1\ (\text{f id})), \pi_2\ (\text{f id}))$
$=$ *by definition of mergeA*
$(\text{encode}\,\phi\ (\pi_1\ (\text{f id})) \vartriangle (\bigtriangleup\ (\pi_2\ (\text{f id}))))$
$=$ *by def of encode*
$\text{P}\,(\text{Embed}\,\phi \circ \text{fromP}\ (\pi_1\ (\text{f id}))) \vartriangle \bigtriangleup\ (\pi_2\ (\text{f id}))$
$=$ *by property of* $\times/\vartriangle$
$\text{P}\,(\text{Embed}\,\phi \times \text{id}) \circ (\pi_1\ (\text{f id}) \vartriangle \bigtriangleup\ (\pi_2\ (\text{f id})))$
$=$ *by definition of mergeA*
$\text{P}\,(\text{Embed}\,\phi \times \text{id}) \circ \bigtriangleup\ (\pi_1\ (\text{f id}), \pi_2\ (\text{f id}))$
$=$ *by contraction of pairs*
$\text{P}\,(\text{Embed}\,\phi \times \text{id}) \circ \bigtriangleup\ (\text{f id})$

- Let $\text{g} = \alpha \circ (\text{split} \times \text{id})$
$\bigtriangleup\ ((\alpha \circ (\text{split} \times \text{id}) \circ \text{f})\ \text{id})$
$=$ *by def of* $\circ$
$\bigtriangleup\ (\alpha\ ((\text{split} \times \text{id})\ (\text{f id})))$
$=$ *by pair expansion*
$\bigtriangleup\ (\alpha\ ((\text{split} \times \text{id})\ (\pi_1\ (\text{f id}), \pi_2\ (\text{f id}))))$
$=$ *by def of* $\times$
$\bigtriangleup\ (\alpha\ ((\text{split}\ (\pi_1\ (\text{f id})), \pi_2\ (\text{f id}))))$
$=$ *by def of split*
$\bigtriangleup\ (\alpha\ ((\pi_1 \circ (\pi_1\ (\text{f id})), \pi_2 \circ (\pi_1\ (\text{f id}))),$
$\pi_2\ (\text{f id})))$
$=$ *by def of assoc*
$\bigtriangleup\ (\pi_1 \circ (\pi_1\ (\text{f id})), (\pi_2 \circ (\pi_1\ (\text{f id})), \pi_2\ (\text{f id})))$
$=$ *by def of mergeA*
$(\pi_1 \circ (\pi_1\ (\text{f id}))) \vartriangle \bigtriangleup\ (\pi_2 \circ (\pi_1\ (\text{f id})), \pi_2\ (\text{f id}))$
$=$ *by def of mergeA*
$(\pi_1 \circ (\pi_1\ (\text{f id}))) \vartriangle ((\pi_2 \circ (\pi_1\ (\text{f id}))) \vartriangle$
$\bigtriangleup\ (\pi_2\ (\text{f id})))$
$=$ *by def of assoc*
$\alpha \circ (((\pi_1 \circ (\pi_1\ (\text{f id}))) \vartriangle (\pi_2 \circ (\pi_1\ (\text{f id})))) \vartriangle$
$\bigtriangleup\ (\pi_2\ (\text{f id})))$
$=$ *by properties of cartesian categories*
$\alpha \circ (\pi_1\ (\text{f id}) \vartriangle \bigtriangleup\ (\pi_2\ (\text{f id})))$
$=$ *by def of mergeA*
$\alpha \circ \bigtriangleup\ (\pi_1\ (\text{f id}), \pi_2\ (\text{f id}))$
$=$ *by contraction of pair*
$\alpha \circ \bigtriangleup\ (\text{f id})$
- Let $\text{g} = (\text{merge} \times \text{id}) \circ \bar{\alpha}$.
$\bigtriangleup\ (((\text{merge} \times \text{id}) \circ \bar{\alpha} \circ \text{f})\ \text{id})$
$=$ *by def of* $\circ$
$\bigtriangleup\ ((\text{merge} \times \text{id})\ (\bar{\alpha}\ (\text{f id})))$
$=$ *by expansion of pairs, def of assoc.*
$\bigtriangleup\ ((\text{merge} \times \text{id})\ ((\pi_1\ (\text{f id}), \pi_1\ (\pi_2\ (\text{f id}))),$
$\pi_2\ (\pi_2\ (\text{f id}))))$
$=$ *by def of* $\times$
$\bigtriangleup\ (\text{merge}\ (\pi_1\ (\text{f id}), \pi_1\ (\pi_2\ (\text{f id}))), \pi_2\ (\pi_2\ (\text{f id})))$
$=$ *by def of merge*
$\bigtriangleup\ ((\pi_1\ (\text{f id}) \vartriangle \pi_1\ (\pi_2\ (\text{f id}))), \pi_2\ (\pi_2\ (\text{f id})))$
$=$ *by def of mergeA*
$(\pi_1\ (\text{f id}) \vartriangle \pi_1\ (\pi_2\ (\text{f id}))) \vartriangle \bigtriangleup\ (\pi_2\ (\pi_2\ (\text{f id})))$
$=$ *by property of* $\vartriangle$*/assoc*
$\bar{\alpha} \circ \pi_1\ (\text{f id}) \vartriangle (\pi_1\ (\pi_2\ (\text{f id})) \vartriangle \bigtriangleup\ (\pi_2\ (\pi_2\ (\text{f id}))))$
$=$ *by def of mergeA*
$\bar{\alpha} \circ \pi_1\ (\text{f id}) \vartriangle \bigtriangleup\ (\pi_2\ (\text{f id}))$
$=$ *by contraction of pairs*
$\bar{\alpha} \circ \pi_1\ (\text{f id}) \vartriangle \bigtriangleup\ (\pi_1\ (\pi_2\ (\text{f id})), \pi_2\ (\pi_2\ (\text{f id})))$
$=$ *by def of mergeA*
$\bar{\alpha} \circ \bigtriangleup\ (\pi_1\ (\text{f id}), \pi_2\ (\text{f id}))$
$=$ *by contraction of pairs*
$\bar{\alpha} \circ \bigtriangleup\ (\text{f id})$
- Let $\text{g} = \lambda\,\text{x} \longrightarrow (\text{unit}, \text{x})$.
$\bigtriangleup\ (((\lambda\,\text{x} \longrightarrow (\text{unit}, \text{x})) \circ \text{f})\ \text{id})$
$=$ *by def of* $\circ$
$\bigtriangleup\ (((\lambda\,\text{x} \longrightarrow (\text{unit}, \text{x})) \circ \text{f})\ \text{id})$
$=$ *by def of unit*
$\bigtriangleup\ (\text{P}\,\varepsilon, \text{f id})$
$=$ *by def of mergeA*
$\varepsilon \vartriangle \bigtriangleup\ (\text{f id})$
$=$ *by property of* ε
$(\varepsilon \vartriangle \text{id}) \circ \bigtriangleup\ (\text{f id})$
$=$ *by property of swap*
$\sigma \circ (\text{id} \vartriangle \varepsilon) \circ \bigtriangleup\ (\text{f id})$
$=$ *by definition of unitor*
$\sigma \circ \rho \circ \bigtriangleup\ (\text{f id})$
- Let $\text{g} = \theta$ be a permutation.
$\bigtriangleup\ ((\theta \circ \text{f})\ \text{id})$
$=$ *by def of* $\circ$
$\bigtriangleup\ (\theta\ (\text{f id}))$
$=$
$\theta \circ \bigtriangleup\ (\text{f id})$
The last step is justified because θ is representable in any symmetric monoidal category. Furthermore, because $\bigtriangleup$ respects the structure of products, it does not matter if θ is applied before or after $\bigtriangleup$.

Recall that the induction hypothesis is that $\bigtriangleup\ (\text{f id})$ is protolinear. This observation alone concludes the argument for the split, merge and unit cases. For the other two cases, it suffices to see that every permutation θ and every generator ϕ is linear, and we have protolinearity for the composition. □

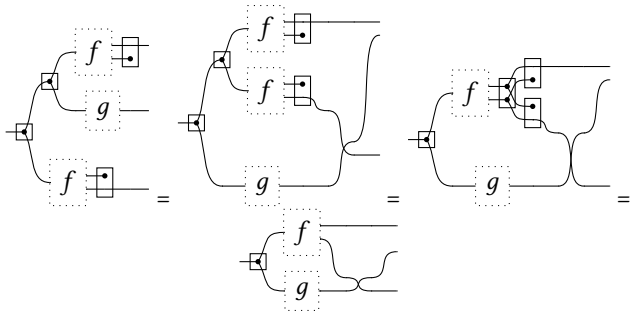


Figure 9. Undoing a split. Two copies of f have been identified. In the first step re-association is performed. Then, f is commuted with δ. Finally duplication and projections are simplified out.

5.3 An Algorithm for reduce

The proof of Theorem 5.6 gives a clear plan for how to implement reduce, namely reducing the form $\bigwedge$ (f id) by induction until we obtain a morphism in SMC form.

However, there are a couple of difficulties to overcome before we actually have a usable algorithm. First, the proof of Lemma 5.7 proceeds by case analysis on the form of the input function ($\mathsf{f} : \mathsf{P\ k\ r\ a} \multimap \bigotimes_i (\mathsf{P\ k\ r\ t}_i)$). But without metaprogramming this form is inaccessible to programs in Haskell: we only have access to the FreeCartesian representation which is *produced* by f id.

Regardless, inspection of the proof of Lemma 5.7 reveals that the bulk of the work, namely undoing split operations, can be done by finding two FreeSMC morphisms of the form $\pi_1 \circ \mathsf{h}$ and $\pi_2 \circ \mathsf{h}$ in the operands of $\bigwedge$, associate them to $(\pi_1 \circ \mathsf{h}) \vartriangle (\pi_2 \circ \mathsf{h})$ and reduce them to h. If we had access to the host language representation, we'd know where these operands were. But we don't: any permutation may be applied to the operands of $\bigwedge$, and therefore an algorithm must start by re-associating them so that $\pi_1 \circ \mathsf{h}$ and $\pi_2 \circ \mathsf{h}$ are connected to the same fork ($\vartriangle$). This step is illustrated in Fig. 9. The process can then continue until all splits have been undone. A complete example involving several such steps is depicted graphically in Fig. 10.

We remark first that the above procedure is terminating, because every transformation reduces the size of the multary merge, as in the proof of Lemma 5.7. The same lemma also tells us that what remains after a reduction step is the computational prefix of the morphism, which is itself protolinear and thus subject to reduction by the same procedure.

Considering all possible re-associations of morphisms and testing for equal prefixes is expensive. Therefore in our implementation we maintain the arguments of $\bigwedge$ as a sorted list of free cartesian morphisms, fs. This ordering is defined lexicographically, considering the components of a composition in computational order (right to left in textual order). Additionally, when comparing $\mathsf{f} \circ \mathsf{g}$ and $\mathsf{f}' \circ \mathsf{g}'$, we ensure

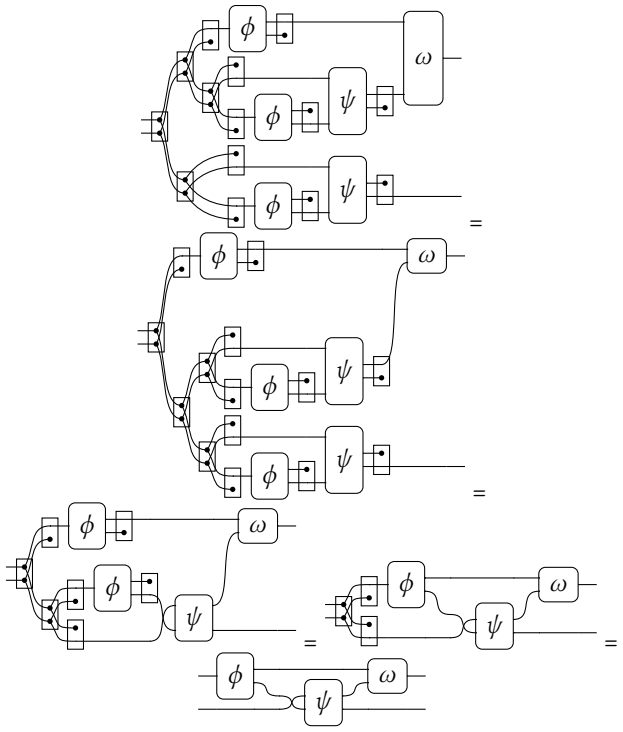


Figure 10. Example of reduction steps.

that neither g nor g' are compositions themselves (otherwise we re-associate compositions). This choice of morphism ordering has two consequences. First, if the morphisms $\pi_1 \circ \mathsf{f}$ and $\pi_2 \circ \mathsf{f}$ are in the sorted list of arguments fs, they must be adjacent to each other: so such a pair is easy to find. Second, f and f' are compared only when g and g' are equal, and this is important in what follows.

One final question remains: how do we arrange to compare g and g' if they are generators (say $\mathsf{g} = \phi$ and $\mathsf{g}' = \psi$)? Do we need to assume a decidable ordering on them? Perhaps surprisingly, the answer is *no*. Indeed, whenever we would need to compare two generators in the reduction procedure, it turns out that they are necessarily equal.

This property can be explained by the conjunction of the following two facts. 1. we compare morphisms only if they have the same source. That is, when we compare $\phi \circ \mathsf{f}$ and $\psi \circ \mathsf{f}'$, we consider the generators ϕ and ψ only if we already know that $\mathsf{f} = \mathsf{f}'$ (thanks to using the lexicographical ordering described above). 2. two generators which have the same source are necessarily equal. This second property is grounded in linearity: the same intermediate result can never be used more than once. Consequently if a generator ϕ is fed an intermediate result x, this same x can never be fed to a *different* generator ψ. (We can end up with two copies of generators in the representation because split makes such copies.)

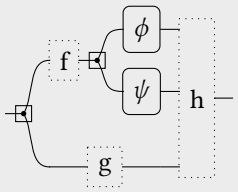
Because we assume that we have two encoded generators ϕ and ψ with the same source, the situation corresponds to them being embedded in a single a morphism of the form

$$\mathsf{h} \circ ((\phi \circ \mathsf{f}) \vartriangle (\psi \circ \mathsf{f}) \vartriangle \mathsf{g}).$$

We start by proving the wanted result, but make a couple of additional assumptions which we discharge later.

Lemma 5.8. *If* $\mathsf{h} \circ ((\phi \circ \mathsf{f}) \vartriangle (\psi \circ \mathsf{f}) \vartriangle \mathsf{g})$ *is pseudolinear and* h *discards neither the output of* ϕ *nor of* ψ*, then* $\phi = \psi$*.*

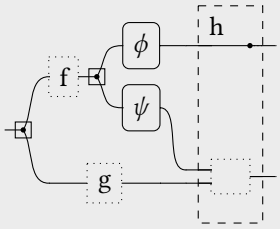
Proof. We have the following equivalence:
$\mathsf{h} \circ ((\phi \circ \mathsf{f}) \vartriangle (\psi \circ \mathsf{f}) \vartriangle \mathsf{g}) = \mathsf{h} \circ (((\phi \times \psi) \circ \delta \circ \mathsf{f}) \vartriangle \mathsf{g})$ So the morphism can be depicted as follows:



But, we also know that it is pseudo-linear, so it can be put in SMC form. In particular, this means that the δ node connecting ϕ and ψ can be eliminated. There are only three ways to reduce this node. We can either 1. assume ϕ=ψ, and then we can apply the rule $(\phi \times \psi) \circ \delta = \delta \circ \phi$, and let further reductions take place; 2. prune away one of (or both) the branches; or 3. assume that there is another copy of ϕ or ψ in g which cause δ commutation and elimination.

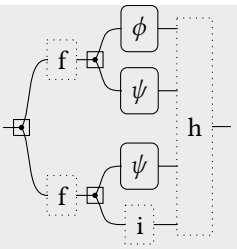
If we can rule out Case 2 and Case 3, then Case 1. must apply, and we have our result: ϕ=ψ.

Case 2. corresponds to one of the branches being equivalent to ε, because some discard occurs inside h. Let us assume without loss of generality that the ϕ branch is the one equivalent to ε. This situation is depicted below:

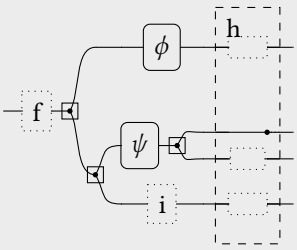


Indeed, the only way that this branch can be pruned is when the output of ϕ is discarded. However, by assumption, we have rejected this situation.

Case 3. can only happen when g is of the form $(\phi \vartriangle \mathsf{i}) \circ \mathsf{f}$ or $(\psi \vartriangle \mathsf{i}) \circ \mathsf{f}$. Let us assume the latter without loss of generality. The situation is then:



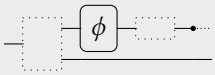
Which reduces to $\mathsf{h} \circ ((\phi \vartriangle (\delta \circ \psi)) \vartriangle \mathsf{i}) \circ \mathsf{f}$ But the only way to reduce the δ node is if one of its branches is connected to ε, as depicted below:



But this could happen only if one of the ψ was discarded to begin with, which is ruled out by assumption. So we can again reject this case.

□

The next step in the argument is to prove that, if any generator is in a decoded morphism, its output never (fully) discarded. That is, the situation depicted in the following diagram cannot occur:



Formally:

Lemma 5.9. *For every* g, h, i *if* $\bigtriangleup (\mathsf{f}\ \mathsf{id}) = \mathsf{h} \circ ((\mathsf{i} \circ \phi) \times \mathsf{id}) \circ \mathsf{g}$*, and* h *is protolinear, then* i *cannot be equivalent to* ε*.*

Proof. The proof is an immediate consequence of the possible ways to construct f— namely, the combinators of our interface. The only way to discard fully a value is via ε, which is itself available only via unit. However, 1. unit applies ε to its input directly and 2. the only construction which places something before another morphism is merge, which places another δ before the whole construction. Consequently ε can only be connected directly to the input of *mergeA (f id)*, never after a generator ϕ. □

To get the desired result, it suffices to put all the pieces together.

Theorem 5.10. *For every function* $\mathsf{f} : \mathsf{P\ k\ r\ a} \multimap \mathsf{P\ k\ r\ b}$*, if* $\bigtriangleup (\mathsf{f}\ \mathsf{id}) = \mathsf{h} \circ ((\phi \circ \mathsf{f}) \vartriangle (\psi \circ \mathsf{f}) \vartriangle \mathsf{g})$*, then* $\phi = \psi$*.*

Proof. We apply Lemma 5.8. The pseudolinearity condition is given by Theorem 5.6, and non-discardability by Lemma 5.9. □

5.4 Haskell Implementation of reduce

In this section we present the main components of the Haskell implementation of the reduce function. We start by showing the underlying data structure which is manipulated by reduce. This data structure is a list of morphisms of type FreeCartesian k a x_i, for varying x_i. (This list corresponds to the arguments of $\bigwedge$.) Because we have to keep track in the type that all these morphisms share the same source object, we need to use a GADT to store them instead of a plain Haskell list:

```
data Merge k a xs where
  (:+) :: FreeCartesian k a x → Merge k a xs
       → Merge k a (Cons x xs)
  Nil  :: Merge k a Null
```

The output of reduce is a morphism whose target object is the monoidal product of the above x_i. So we need to encode the product of a list of types, as a type family:

```
type family Prod (xs :: [Type]) where
  Prod Null = ()
  Prod (Cons x ys) = x ⊗ Prod ys
```

As explained above, lists of morphisms will be sorted according to the lexicographical order on FreeCartesian k a. To keep lists in sorted order, we will need a function to merge them while preserving the order. Even though this kind of function is entirely standard, our version must only return the resulting merged list, but it must also keep track of the permutations and re-associations which it applies. Indeed, this permutation is necessary for the purpose of the algorithm, because overall the meaning of the morphism must remain the same: the composition of the sorted merge operation and the permutations is the identity. Because we do not know, from types only, the ordering of the resulting list and hence its type, we must quantify existentially over it. Because Haskell does not support native existential types, we use a CPS encoding to define the append function:

```
appendSorted :: Merge cat a xs → Merge cat a ys →
                (∀ zs. FreeSMC cat (Prod zs)
                                   (Prod xs ⊗ Prod ys) →
                  Merge cat a zs → k) → k
appendSorted Nil       ys       k = k (σ ∘ ρ) ys
appendSorted xs        Nil      k = k      ρ  xs
appendSorted (x :+ xs) (y :+ ys) k =
  case compareMorphisms x y of
    GT → appendSorted (x :+ xs) ys $ λ a zs →
           k (α ∘ (σ × id) ∘ ᾱ ∘ (id × a)) (y :+ zs)
    _  → appendSorted xs (y :+ ys) $ λ a zs →
           k (          ᾱ ∘ (id × a)) (x :+ zs)
```

Like appendSorted, the rest of the functions must record permutations which they might apply, and thus are written in the same style, with existentials encoded in CPS. In fact, when the input sorted list of morphisms is empty, the accumulated permutation contains the result morphism in FreeSMC form.

The purpose of the next function is to expose forks (△) as a sorted list of morphisms to merge (of type Merge). Additionally, it shifts embedded morphisms (and ε, which when merged is a no-op) to the accumulated result.

```
expose :: Cat cat a b →
      (∀ x. FreeSMC cat (Prod x) b →
       Merge cat a x → k) → k
expose (f₁ :△: f₂) k        = expose f₁ $ λ g₁ fs₁ →
                              expose f₂ $ λ g₂ fs₂ →
                              appendSorted fs₁ fs₂ $ λ g fs →
                              k ((g₁ × g₂) ∘ g) fs
expose (Embed φ :<: f) k    = expose f $ λ g fs →
                              k (FreeSMC.Embed φ ∘ g) fs
expose (E :<: _) k          = k id Nil
expose x k                  = k ρ̄ (x :+ Nil)
```

To finish we show the code to undo a split. Even though it is somewhat obscured by the necessary accumulation of result morphisms, its purpose is simple: searching the sorted list for a pair $\pi_1 \circ f$ and $\pi_2 \circ f$ and apply the appropriate reduction.

```
reduceStep :: Merge cat a xs →
        (∀ zs. FreeSMC cat (Prod zs) (Prod xs) →
          Merge cat a zs → k) → k
reduceStep ((P₁ :<: f₁) :+ (P₂ :<: f₂) :+ rest) k
  | EQ ← compareMorphisms f₁ f₂ =
  expose f₁                $ λ g f' →
  appendSorted f' rest $ λ g' rest' →
  k (α ∘ (g × id) ∘ g') rest'
reduceStep (f :+ rest) k =
  reduceStep rest $ λ g rest' →
  appendSorted (f :+ Nil) rest' $ λ g' rest'' →
  k ((ρ̄ × g) ∘ g') rest''
```

6 Discussion and Related Work

6.1 Dynamically Checking for Linearity

Could we implement a variant of our API and implementation which performs linearity checks at runtime, rather than relying on Haskell to perform them? This sounds reasonable: after all we already construct a representation of decoded morphisms, and we can run a protolinearity (Definition 5.5) check on it. This could be done, but only if generators are equipped with a decidable equality. Indeed, consider the morphism $(\pi_1 \circ \phi) \triangle (\pi_2 \circ \psi)$. It is protolinear if ϕ=ψ, but not otherwise. The tradeoff is simple to express: one either needs static linearity checks or a dynamic equality check on generators (but not both). However, if one would choose dynamic equality checks, it may be more sensible to evaluate to cartesian categories instead, as discussed in Section 6.2.

6.2 Evaluating to Cartesian Categories

Our technique can be adapted to cartesian structures (instead of monoidal symmetric ones). To do so one shall 1. retain the encoding of ports as morphisms from an abstract object r: P k r a = FreeCartesian k r a, 2. relax the requirement to work

with linear functions, and 3. drop the projection from free cartesian to free monoidal structures in the implementation of decode. We must however underline that such a technique places δ at the earliest points in the morphisms, thus generators are duplicated every time their output is split. This behaviour follows cartesian laws to the letter: indeed they stipulate that such duplication has no effect: $(f \times f) \circ \delta = \delta \circ f$. However, categories which make both sides of the above equation equivalent in all respects are rare. For example the presence of effects tend to break the property. For example, a Kleisli category is cartesian only if the embedded effects are commutative and idempotent. In particular if one takes runtime costs into account, the equivalence vanishes. Worse, in the presence of other optimisations, one cannot tell *a priori* which side of the equation has the lowest cost: it may be beneficial to have a single instance of f so that work is not duplicated, but it may just as well be more beneficial to have two instances, so that for example they can fuse with whatever follows in the computation. Indeed if the output of f is large, following it with δ may require storing (parts of) it, whereas each copy of f may be followed by a function which only require f to be ran lazily, not requiring any storage. In general, programmers must decide for themselves if it is best to place f before or after δ. Thus the SMC approach, which we follow, is to ask the programmer to place δ *explicitly*, using copy from Section 3.[4] We regard this approach to be the most appropriate when there is a significant difference between the left- and the right-hand side of the above equation. Another possibility would be to use a decidable check over generators (as discussed in Section 6.1) and enforce that all applications of a generator to the same input are realised as a single generator in the resulting morphism. For example, one can use identity in the source code (of the host language) as generator equality. Then, each occurrence in the source is mapped one-to-one with its occurrences in the representation as a (cartesian) morphism. This sort of source-code identity is available when one has access to the representation of the source code (see Section 6.4), or by using any approach to observable sharing (see Section 6.3).

6.3 Observable Sharing

One way to recover representations from embedded DSLs is to leverage observable sharing techniques. Gill [8] provides a review of the possible approaches, but in short, one uses unique names equipped with testable equality for what we call here generators. Explicit unique names can be provided directly by the programmer, or generated using a state monad. Alternatively, testable equality can be implemented by pointer equality. The version of Claessen and Sands [6] is native, but it breaks referential transparency. The version of Peyton Jones et al. [18] preserves referential transparency, but resides in the catch-all IO monad.

Turning the problem on its head, we can see our approach as a principled solution to observable sharing. Essentially, forcing the programmer to be explicit about duplication means that no implicit sharing needs to be recovered, and therefore EDSL backends (such as those presented in Section 4) need not deal with it.

6.4 Compiling to Categories

Elliott [7] advertises a compiler plugin which translates a source code representation to a categorical representation. This plugin is close in purpose to what we propose here. The first obvious difference is that our solution is entirely programmed within Haskell, while Elliott's acts at the level of the compiler. This makes our approach much less tied to a particular implementation, and we even expect it to be portable to other languages with linear types. In return, it demands paying an extra cost at runtime.

There are more fundamental differences however: because the input of the plugin is Haskell source code, it is forced to target cartesian closed categories, even though most of Elliott's applications naturally reside at the simple cartesian level (keeping in mind the *caveat* discussed in Section 6.2). This forces one to provide cartesian closed instances for all applications, or add a translation layer from cartesian closed to just cartesian categories. Our approach avoids any of those complications, but Valliappan and Russo [22] provide a detailed study of the alternative.

In fact, the present work has much synergy with Elliott's: all his examples are supported by our technique, out of the box, and we recommend consulting them for a broader view of the applications of categorical approaches. Accordingly, in Section 4 we have focused on the stones left unturned by Elliott. In particular the applications to quantum gates is out of reach when one targets cartesian closed categories.

6.5 A Practical Type Theory for SMCs

Shulman [20] proposes a type theory for SMCs. The motivation is different than ours: Shulman wants to provide set-like reasoning on SMC morphisms to mathematicians, whereas we are providing a notation to describe SMC morphisms in a programming language. Shulman's is a dedicated language while ours is embedded in Haskell. The means are rather different too: Shulman's type theory doesn't require giving a meaning to ports (Shulman calls ports "terms") instead the semantic is given globally over an entire judgement. We give a local semantics by giving a meaning to ports. This difference follows from our implementing the port interface within Haskell, rather than using metaprogramming, as described in Section 6.4.

Nevertheless, the end product, as far as the user is concerned, is pretty similar: one writes expressions on ports that

[4] Indeed, duplications which we consider here are coming from user code, and they are disjoint from those that we insert in the intermediate free cartesian representations discussed in Section 5. In fact, there is no interaction between the two.

one then needs to combine together to form a legal expression. This convergence suggests that there may be value in a further investigation of the mathematical structure of ports.

6.6 Quantification over r

Another minor possible improvement in the API (Section 3) would be to remove the variable r in the type P k r a. Such locally quantified variables are used to ensure that two instances of an EDSL are not mixed together. For instance, Launchbury and Peyton Jones [10] use them to capture the identity of state threads. However, in our case, this role is already fulfilled by the use of linear types. Indeed, if an initial value type P k r a is introduced by an instance of decode, linearity checks already prevent it from occuring free in another instance of decode. The same property holds for values derived from it (using split, merge, etc.). We leave a proof of this fact to future work. Besides, even though the implementation does not strictly need the variable r, our proof of its correctness does, therefore we have not explored this route further.

7 Conclusion

When defining an embedded domain specific language, there is often a tension between making the syntax (API) convenient for the user, and making the implementation simple. In particular, how to compose objects is an important choice in the design space. Using explicit names for intermediate computations is often most convenient for the user, but can be hard to support by the implementation. In this paper we have shown a way to bridge the gap between the convenience of lambda notation on the user-facing side with the convenience of categorical combinators on the implementation side. The price to pay is linearity: the user must make duplication and discarding of values explicit.

Indeed, our technique is grounded in the equivalence between symmetric monoidal categories and linear functions. While this equivalence is well known, we have pushed the state of the art by showing that linear functions can *compute their own representation* in a symmetric monoidal category.

Our technique has several positive aspects: it is usable in practice in a wide range of contexts; it is comprised of a small interface and reasonably short implementation; and it does not depend on any special-purpose compiler modification, nor on metaprogramming. As such, in the context of Haskell, it has the potential to displace the arrow notation as a standard means to represent computations whose static structure is accessible.

In general, we think that this paper provides suitable means to work with commutative effects in functional languages. Commutative effects are numerous (environment, supply of unique names, random number generation, etc.), and proper support for them has been recognised as a challenge for a long time, for example by Peyton Jones [16, challenge 2, slide 38]. This paper provides evidence that SMCs constitute the right abstraction for commutative effects, and that linear types are key to providing a convenient notation for them.

Acknowledgments

We warmly thank James Haydon and Georgios Karachalias as well as anonymous reviewers for feedback on previous versions of this paper. Jean-Philippe Bernardy is supported by grant 2014-39 from the Swedish Research Council, which funds the Centre for Linguistic Theory and Studies in Probability (CLASP) in the Department of Philosophy, Linguistics, and Theory of Science at the University of Gothenburg.

References

[1] Michael Barr and Charles Wells. 1999. *Category Theory for Computing Science* (third ed.). Prentice Hall.

[2] P. N. Benton. 1995. A mixed linear and non-linear logic: Proofs, terms and models. In *Computer Science Logic*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 121–135.

[3] Jean-Philippe Bernardy, Arnaud Spiwack, Mathieu Boespflug, Ryan Newton, and Simon Peyton Jones. 2018. Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages* 2 (2018). Issue POPL.

[4] Guillaume Boisseau and Jeremy Gibbons. 2018. What You Needa Know about Yoneda: Profunctor Optics and the Yoneda Lemma (Functional Pearl). *Proc. ACM Program. Lang.* 2, ICFP, Article 84 (jul 2018), 27 pages. https://doi.org/10.1145/3236779

[5] Edwin Brady. 2020. Idris 2: Quantitative Type Theory in Action. https://www.type-driven.org.uk/edwinb/papers/idris2.pdf

[6] Koen Claessen and David Sands. 1999. Observable Sharing for Functional Circuit Description. In *Advances in Computing Science - ASIAN'99, 5th Asian Computing Science Conference, Phuket, Thailand, December 10-12, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1742)*, P. S. Thiagarajan and Roland H. C. Yap (Eds.). Springer, 62–73. https://doi.org/10.1007/3-540-46674-6_7

[7] Conal Elliott. 2017. Compiling to categories. *Proc. ACM Program. Lang.* 1, ICFP (2017), 27:1–27:27. https://doi.org/10.1145/3110271

[8] Andy Gill. 2009. Type-safe observable sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, Stephanie Weirich (Ed.). ACM, 117–128. https://doi.org/10.1145/1596638.1596653

[9] John Hughes. 2000. Generalising monads to arrows. *Sci. Comput. Program.* 37, 1-3 (2000), 67–111. https://doi.org/10.1016/S0167-6423(99)00023-4

[10] John Launchbury and Simon Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, USA, June 20-24, 1994*, Vivek Sarkar, Barbara G. Ryder, and Mary Lou Soffa (Eds.). ACM, 24–35. https://doi.org/10.1145/178243.178246

[11] Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access. *SIGPLAN Not.* 49, 9 (Aug. 2014), 325–337. https://doi.org/10.1145/2692915.2628144

[12] Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's do-notation into applicative operations. In *Proceedings of the 9th International Symposium on Haskell,*

Haskell 2016, Nara, Japan, September 22-23, 2016, Geoffrey Mainland (Ed.). ACM, 92–104. https://doi.org/10.1145/2976002.2976007

[13] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *PACMPL* 3, ICFP (2019), 110:1–110:30. https://doi.org/10.1145/3341714

[14] Yves Parès, Jean-Philippe Bernardy, and Richard A. Eisenberg. 2020. Algebraic Effects with Tasks Build Composable Data Workflows. In *Proceedings of the Haskell Symposium.*

[15] Ross Paterson. 2001. A new notation for arrows. *ACM SIGPLAN Notices* 36, 10 (2001), 229–240.

[16] Simon Peyton Jones. 2003. Wearing the hair shirt: a retrospective on Haskell (2003). https://www.microsoft.com/en-us/research/publication/wearing-hair-shirt-retrospective-haskell-2003/ invited talk at POPL 2003.

[17] Simon Peyton Jones, Andrew D. Gordon, and Sigbjörn Finne. 1996. Concurrent Haskell. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 295–308. https://doi.org/10.1145/237721.237794

[18] Simon Peyton Jones, Simon Marlow, and Conal Elliott. 1999. Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell. In *Implementation of Functional Languages, 11th International Workshop, IFL'99, Lochem, The Netherlands, September 7-10, 1999, Selected Papers (Lecture Notes in Computer Science, Vol. 1868)*, Pieter W. M. Koopman and Chris Clack (Eds.). Springer, 37–58. https://doi.org/10.1007/10722298_3

[19] P. Selinger. 2011. *A Survey of Graphical Languages for Monoidal Categories.* Springer Berlin Heidelberg, Berlin, Heidelberg, 289–355. https://doi.org/10.1007/978-3-642-12821-9_4

[20] Michael Shulman. 2019. A practical type theory for symmetric monoidal categories. *arXiv preprint arXiv:1911.00818* (2019). https://arxiv.org/abs/1911.00818

[21] Manfred E Szabo. 1978. *Algebra of proofs.* Elsevier.

[22] Nachiappan Valliappan and Alejandro Russo. 2019. Exponential Elimination for Bicartesian Closed Categorical Combinators. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming* (Porto, Portugal) *(PPDP '19).* Association for Computing Machinery, New York, NY, USA, Article 20, 13 pages. https://doi.org/10.1145/3354166.3354185