Procontainers for idioms, arrows and monads

Exequiel Rivas
Tallinn University of Technology, Tallinn, Estonia
exequiel.rivas@taltech.ee

Containers, or polynomial functors, are a popular class of functors that is closed under a variety of operations, including coproducts, products, Day convolution and composition. Idioms and monads, two extremely successful interfaces for capturing computational effects in functional languages, are based on these last two operations between functors. However, a third popular interface of computational effects is missing from the picture: arrows, which are usually understood in terms of *pro*functors instead of functors. In this article we introduce a notion of *pro*container, which is a class of profunctors also closed under the operations of interest. We demonstrate how this notion allows us to express the well-known connections between these three different interfaces for computational effects, when restricted to containers.

1 Introduction

The motivation for this work comes from the relation between functors and profunctors that appeared in functional programming, in particular, in the treatment of computational effects. The most notable interfaces for expressing computational effects inside a functional programming language are monads, idioms (or applicative functors), and arrows. Some years ago, Lindley, Yallop and Wadler [10] provided a detailed study of the expressive power relating these three interfaces by syntactical means. Their results can be summed up with a diagram

$$\text{Idioms} \hookrightarrow \text{Arrows} \hookrightarrow \text{Monads}$$

together with the equations

$$\text{Idiom} = \text{Arrow} + (A \rightsquigarrow B \cong 1 \rightsquigarrow (A \to B)), \tag{1}$$

$$\text{Monad} = \text{Arrow} + (A \rightsquigarrow B \cong A \to (1 \rightsquigarrow B)). \tag{2}$$

stating that monads and idioms are special cases of arrows that satisfy particular equations. Some years after, Rivas [18] showed the same results in a more semantical way, by using the fact that these three interfaces are built on categorical concepts, the first two on top of the notion of *functor*, and arrows built on top of *profunctors* (or more precisely, *strong profunctors*). This work, however, is done by restricting to functors in the categories $[\mathbb{F}, \mathbf{Set}]$ and $[\mathbb{F}^{\mathrm{op}} \times \mathbb{F}, \mathbf{Set}]_{\mathrm{s}}$, where $\mathbb{F}$ represents the category of finite sets. The maps corresponding to the diagram above are obtained by involved formulas in terms of ends and coends. Instead, we might consider exploring what happens when we restrict ourselves to other families of functors, which could potentially offer simpler terms for expressing the mappings between interfaces. One such class of functors that meets these criteria is known as *containers*, and this paper is a development from this motivation.

In the rest of this section, in order to explain our proposal and present contributions, we give an overview of these interfaces and a brief description of containers (which is expanded later in Section 2)

Submitted to:
MSFP 2024

1.1 Computational effects

Naively, a computer program is like a mathematical function: it takes some input, and returns some output, it is a functional relation. However, computers can *do* much more. Computational effects refer to the "things" that a program can do. For example, a computer program can print to a terminal, or read from an Internet socket, or behave non-deterministically depending on some external decision. When studying semantics of computer programs, we need to model these effects, and different mathematical structures have emerged to explain them. At the same time, these mathematical structures provide the functional programmer with an *interface* to express effects and structure code, and this is why they have been popularized in the functional programming community. Three such interfaces are those that interest us in this work: *monads*, *idioms* (or *applicative functors*), and *arrows*.

When presenting mathematical concepts, we will restrict ourselves to the case that the ambient category is **Set**. We use Agda to express how the interfaces could look inside a programming language, and we will not represent here the equational laws (as in programming languages like Haskell or OCaml, these cannot be normally expressed. Our presentation is similar to Agda standard library's *raw* versions).

Originally, monads were used by Moggi [12, 13] for capturing the semantics of programming languages with effects, and not much after, Wadler [22] observed that effectful functional programs could be structured following an interface provided by modeling a monad inside the language. This construction builds on the notion of functor, so we begin by representing a functor in Agda as a record

```
record Functor (F : Set → Set) : Set₁ where
  field map : ∀ {X Y : Set} → (X → Y) → F X → F Y
```

denoting a functorial action for a type constructor. For a monad, we extend this record with two polymorphic functions which capture the monad structure:

```
record Monad F {{_ : Functor F}} : Set₁ where
  field η : ∀ {A : Set} → A → F A
        μ : ∀ {A : Set} → F (F A) → F A
  open Functor {{...}} public
  _⟩⟩=_ : ∀ {A B} → F A → (A → F B) → F B
  _⟩⟩=_ v f = μ (map f v)
```

Under the interpretation that FX represents computations of type X, η is used to lift (pure) values to computations, and μ is used to compose computations, although in practice generally the equivalent combinator $\rangle\rangle=$ is used.

Some years later, McBride and Patterson [11] introduced *idioms* as another interface which can help to capture computational effects, where this time computations cannot depend on previous results. As with a monad, we have a functor F, together with functions:

```
record Idiom F {{_ : Functor F}} : Set₁ where
  field pure : ∀ {X : Set} → X → F X
        app : ∀ {X Y : Set} → F (X → Y) → F X → F Y
  open Functor {{...}} public
  _‖_ : ∀ {A B} → F A → F B → F (A × B)
  v ‖ w = app (map _,_ v) w
```

Now pure values are lifted to computations using pure, and computations are combined using app. Categorically speaking, idioms are characterized as *(strong) lax monoidal functors*, where we can derive monoidality as in ‖.

In between monads and idioms, Hughes [7] introduced another interface for expressing computations, aiming to generalize monads. However, this time, it was not assumed that we had a functor, but instead a type constructor which would take two parameters and act contravariantly/covariantly on each of them, which is captured in the notion of *profunctor*:

```
record Profunctor (P : Set → Set → Set) : Set₁ where
  field dimap : ∀ {X Y C D} → (Y → X) → (C → D) → P X C → P Y D
```

This record captures what in category theory would be a profunctor $\mathbf{Set} \nrightarrow \mathbf{Set}$, i.e. a functor $\mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \to \mathbf{Set}$. The structure introduced by Hughes to capture effects is called *arrow*, and these endow a profunctor with the following interface combinators that allow to combine computations:

```
record Arrow P {{_ : Profunctor P}} : Set₁ where
  field arr : ∀ {X Y : Set} → (X → Y) → P X Y
        _⋙_ : ∀ {X Y Z : Set} → P X Y → P Y Z → P X Z
        first : ∀ {X Y Z : Set} → P X Y → P (X × Z) (Y × Z)
```

The intuition is that a term $c : P\ X\ Y$ is a computation with input X and output Y. Trivial computations are pure functions obtained using arr. Computations can be sequencialized using ⋙, and we can pass around unmodified data using first.

As these interfaces were introduced, the differences between them were studied. An immediate fact is that both monads and idioms can be turned into arrows. To see this, we can see two ways in which a functor can be turned into a profunctor:

```
Kleisli : (Set → Set) → Set → Set → Set
Kleisli F X Y = X → F Y

instance Kleisli-profunctor : ∀ {F} {{_ : Functor F}} → Profunctor (Kleisli F)
```

and

```
Cayley : (Set → Set) → Set → Set → Set
Cayley F X Y = F (X → Y)

instance Cayley-profunctor : ∀ {F} {{_ : Functor F}} → Profunctor (Cayley F)
```

Then, instances translating a monad to an arrow and an idiom to an arrow are:

```
instance
  Kleisli-monad : ∀ {F} {{_ : Functor F}} {{_ : Monad F}} → Arrow _ {{Kleisli-profunctor}}
  Kleisli-monad = let open Monad {{...}} in
    record { arr = λ f x → η (f x) ;
             _⋙_ = λ f g x → μ (map g (f x)) ;
             first = λ { f (x , z) → map (λ y → y , z) (f x) } }

instance
  Cayley-idiom : ∀ {F} {{_ : Functor F}} {{_ : Idiom F}} → Arrow _ {{Cayley-profunctor}}
  Cayley-idiom = let open Idiom {{...}} in
    record { arr = pure ;
             _⋙_ = λ v → app (map (λ f g x → g (f x)) v) ;
             first = λ { v → map (λ { f (x , z) → f x , z }) v } }
```

These transformations will guide our development of a notion of procontainers.

1.2 Containers

In computer science, *containers* [1] were introduced in the quest for a characterization of data-types that is closed under certain usual operations. The idea is that such data-type constructors can be provided giving: a set of *shapes*, and a *position* function that assigns a set to each shape. A typical example of a container is the *list functor*, in which we take $\mathbb{N}$ as the set of shapes ("a list of length ..."), and for each shape $n : \mathbb{N}$, the positions are the finite sets $\{0, \ldots, n-1\}$ (each list of length n has n positions, corresponding to its elements). On the mathematical side, containers were treated extensively, named *polynomial functors*, and a number of connections to different concepts in mathematics such as *operads* were made [9, 5], recently capturing the attention of the *applied category theory* school [19].

Both monads and idioms have a functor in its underlying construction, and we can wonder when this functor is a container. Indeed, monads and idioms can be seen as monoids under operations closed in containers.

1.3 Our proposal

How do arrows fit in the picture when we restrict computational effects to containers? As explained before, arrows build on profunctors instead of functors, so a natural point of departure is to wonder what would be an appropriate notion of *polynomial profunctor*, or as we will call them, *procontainer*. After we have such a notion, how do the transformations we saw between monads, idioms and arrows work between containers and procontainers? We will explore these points in the article.

We will give a type theory representation of the concepts using Agda (in the spirit of containers), but also a mathematical presentation of some of these concepts as well (in the spirit of the polynomial functors as bundles). We will work with the category of sets and functions, which we denote as **Set**.

The rest of the article is structured as follows. In Section 2, we do a review on containers, while in Section 3 we present a definition for procontainers. Later, in Section 4, we show how some procontainers operations can be defined. In Section 5, we highlight some relations between containers and procontainers, highlighting the parallel with the transformations seen in this introduction, and see a similar pattern for Dirichlet functors in Section 6. In Section 7, we propose some discussion about possible generalizations, as well as limitations of our proposed notion of procontainer. Finally, in Section 8, we conclude.

1.4 Contributions

We highlight the major contributions of this article:

1. We introduce a definition of procontainers, both in Agda code and in mathematical terms as composable morphisms in the category **Set**;

2. We show that procontainers are closed under operations of interest, particularly that of Bénabou's tensor (horizontal composition of profunctors);

3. We demonstrate how procontainers are related to containers (and Dirichlet functors), obtaining particular cases of results known for computational effects.

2 Containers, or polynomial functors

A (non-indexed) polynomial diagram is a morphism in **Set**

$$E \xrightarrow{p} B$$

seen as a bundle. We see the object B as an object of shapes, and each *fiber* E_b is seen as the positions corresponding to the shape b. We will also provide an Agda version of the constructions, which allows us to stay closer to type theory, the setting where much of the container theory have been developed. For Agda, we can take a more direct approach, and directly encode a container as a set of shapes together with a function assigning to each shape a set of positions:

```
record Container : Set₁ where
  constructor _▷_
  field Shape : Set
        Position : Shape → Set
```

Containers can be realized as functors, which we code with the help of the Σ-types former Σ:

```
⟦_⟧ᶜ : Container → Set → Set
⟦ S ▷ P ⟧ᶜ Y = Σ[ s ∈ S ] (P s → Y)
```

As this is a *functor*, we also need to describe what is the action on maps, but this is straightforward:

```
mapᶜ : ∀ {C : Container} {X Y : Set} → (X → Y) → ⟦ C ⟧ᶜ X → ⟦ C ⟧ᶜ Y
mapᶜ f (s , k) = s , (λ x → f (k x))
```

Mathematically, we can write this realization as a composition of functors:

$$\mathbf{Set} \xrightarrow{\Delta_{!_E}} \mathbf{Set}/E \xrightarrow{\Pi_p} \mathbf{Set}/B \xrightarrow{\Sigma_{!_B}} \mathbf{Set}$$

Here, $!_A : A \to 1$ denotes the unique morphism from A to the terminal object, and, given a function $f : A \to B$, the functor $\Delta_f : \mathbf{Set}/B \to \mathbf{Set}/A$ applied on an object $h : X \to B$ of the slice $\mathbf{Set}/B$ is the pullback on Figure 1a. For each $f : A \to B$, we have that $\Sigma_f \dashv \Delta_f$ for $\Sigma_f(g : X \to A) = f \circ g$. In addition, Δ_f has a right adjoint Π_f, which is equivalent to the fact that **Set** is a locally Cartesian closed category (LCCC).

The morphisms between containers are defined as follows, where shapes are mapped in a forward direction, while positions are mapped in a backward direction:

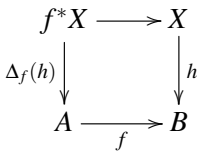
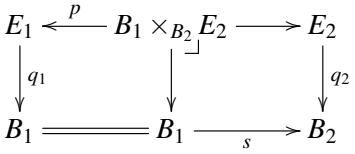


Figure 1: (a) Pullback defining Δ_f.



(b) Container morphism.

```
record _⇒ᶜ_ (C₁ : Container) (C₂ : Container) : Set where
  constructor _▷_
  field shapemap : Shape C₁ → Shape C₂
        positionmap : ∀ s → Position C₂ (shapemap s) → Position C₁ s
```

Just as containers can be realized as functors, the morphisms between containers can be realized as natural transformation between the corresponding realized functors:

```
⟨⟨_⟩⟩ᶜ : ∀ {C₁ C₂} (f : C₁ ⇒ᶜ C₂) → ∀ {X} → ⟦ C₁ ⟧ᶜ X → ⟦ C₂ ⟧ᶜ X
⟨⟨_⟩⟩ᶜ (sm ▷ pm) = λ { (s , k) → sm s , λ x → k (pm s x) }
```

In terms of bundles, a morphism from $q_1 : E_1 \to B_1$ to $q_2 : E_2 \to B_2$ is given by the data (s, p) such that the diagram in Figure 1b commutes.

Interestingly, there is an alternative realization of a container as a functor. Spivak and Myers have studied these under the name of Dirichlet functors [14]. Renaming containers to Dirichlet,

```
Dirichlet = Container
```

we can show how they are realized as contravariant functors:

```
⟦_⟧ᴰ : Dirichlet → Set → Set
⟦ S ▷ P ⟧ᴰ Y = Σ[ s ∈ S ] (Y → P s)
```

We can think of this realization as a sum of contravariant representables. In the code definition above, the *contravariance* aspect is not visible. We see contravariance in its action on maps:

```
mapᴰ : ∀ {D : Dirichlet} {X Y : Set} → (X → Y) → ⟦ D ⟧ᴰ Y → ⟦ D ⟧ᴰ X
mapᴰ f (s , k) = s , (λ x → k (f x))
```

When we consider this construction in a fiberwise manner, we can see that now the realization is as follows:

$$\mathbf{Set}^{\mathrm{op}} \xrightarrow{\Delta^{\mathrm{op}}_{!_B}} (\mathbf{Set}/B)^{\mathrm{op}} \xrightarrow{[-,p]_{/B}} \mathbf{Set}/B \xrightarrow{\Sigma_{!_B}} \mathbf{Set}$$

where $[-,+]_{/B}$ is the internal hom for the slice category $\mathbf{Set}/B$ (guaranteed to exist as **Set** is an LCCC).

This alternative realization comes paired together with an alternative notion of morphism between containers, which works in a forward direction for the positions:

```
record _⇒ᴰ_ (D₁ : Dirichlet) (D₂ : Dirichlet) : Set where
  constructor _▷_
  field shapemap : Shape D₁ → Shape D₂
        positionmap : ∀ {s} → Position D₁ s → Position D₂ (shapemap s)
```

These morphisms are sometimes called *charts*. In terms of bundles, a morphism from $q_1 : E_1 \to B_1$ to $q_2 : E_2 \to B_2$ is given by maps (s, p) making the regular diagram commute: $s \cdot q_1 = q_2 \cdot p$. This view gives a characterization of the category of Dirichlet functors as the category of arrows on **Set** [14].

2.0.1 Operations on containers

An interesting property of containers is that they are closed under multiple operations such as products, coproducts, composition, Day convolution, etc. In this article we will focus in two of them: composition and Day convolution.

Since containers are realized as endofunctors **Set** → **Set**, which we can compose, a natural operation to perform between two containers is to compose them, which is defined for containers as:

$$(S \triangleright P) \circ^{\mathbf{C}} (T \triangleright Q) = [\![\ (S \triangleright P)\ ]\!]^{\mathbf{C}}\ T \triangleright \lambda\ \{\ (s\ ,\ v) \to \Sigma[\ p \in P\ s\ ]\ Q\ (v\ p)\ \}$$

Naturally, we have that the realization of composition of containers is the composition of the realizations. Together with the identity container, $\mathrm{Id} = 1 \triangleright \lambda\_.1$, composition forms a monoidal structure on **Cont**. An interesting point is that the monoids for this monoidal structure are realized as monads.

Another interesting operation to perform between containers is given by *Day convolution*. We can define it as follows:

$$(S \triangleright P) \star^{\mathbf{C}} (T \triangleright Q) = (S \times T) \triangleright \lambda\ \{\ (s\ ,\ t) \to P\ s \times Q\ t\ \}$$

Here, the convolution is performed with respect to products in the category **Set**. More generally, for any symmetric monoidal product on **Set**, there is a corresponding symmetric monoidal structure on **Cont** (see [20]). Again, Day convolution together with the identity container form a monoidal structure on **Cont**, and monoids for this monoidal structure endow functors realizing the underlying containers of monoids with the structure of a lax monoidal functor, i.e. an idiom.

3 Procontainers, or polynomial profunctors

We now move to the definition of procontainers. As we remarked in the introduction, our inspiration for this is to look into the transformations between functors and profunctors used in functional programming. There are in principle many options for doing this, we pick a particular one because the closure operations (Section 4) and its connection to containers (Section 5), we discuss some of the consequences of this choice later in Section 7.

We will define *procontainers* using the following record:

```
record ProContainer : Set₁ where
  constructor _▷_▷_
  field Shape : Set
        Position⁺ : Shape → Set
        Position⁻ : (s : Shape) → (s⁺ : Position⁺ s) → Set
```

As before, there are a number of possible shapes, but now for each shape, instead of having a single set, we will now have an indexed set, which is represented by a set of indices and a set for each element in that index. It is important now to see what is the interpretation we have in mind for this data. For that, we show how to realize a procontainer as a profunctor. The first step is to show its action on objects:

```
⟦_⟧ : ProContainer → Set → Set → Set
⟦ S ▷ P⁺ ▷ P⁻ ⟧ X Y = Σ[ s ∈ S ] (X → Σ[ p⁺ ∈ P⁺ s ] (P⁻ s p⁺ → Y))
```

This interpretation will be contravariant in the first argument, and covariant in the second one. This can be seen in its realization as natural transformations in two arguments:

$$\begin{aligned}&\mathsf{map} : \forall\ \{PC\}\ \{X\ X'\}\ \{Y\ Y'\} \to \\ &\qquad (X' \to X) \to (Y \to Y') \to [\![\ PC\ ]\!]\ X\ Y \to [\![\ PC\ ]\!]\ X'\ Y' \\ &\mathsf{map}\ f\ g\ (t\ ,\ h) = t\ ,\ \lambda\ x' \to (\mathsf{proj}_1\ (h\ (f\ x'))))\ ,\ \lambda\ z \to g\ (\mathsf{proj}_2\ (h\ (f\ x'))\ z)\end{aligned}$$

The bundle point of view can be captured as follows. A polynomial profunctor diagram is

$$F \xrightarrow{q} E \xrightarrow{p} B$$

Informally, we will call the morphism q the *head*, and p the *tail*. From the bundles point of view, we can construct the profunctor as follows, which is a combination of a Dirichlet functor and container representation:

$$\mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \xrightarrow{\mathrm{id}\times\Delta_{!_F}} \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set}/F \xrightarrow{\mathrm{id}\times\Pi_q} \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set}/E \xrightarrow{\mathrm{id}\times\Sigma_p} \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set}/B$$

$$\xrightarrow{\Delta^{\mathrm{op}}_{!_B}\times\mathrm{id}} (\mathbf{Set}/B)^{\mathrm{op}} \times \mathbf{Set}/B \xrightarrow{[-,+]_{/B}} \mathbf{Set}/B \xrightarrow{\Sigma_{!_B}} \mathbf{Set}$$

A profunctor can come with a strength, which is the generalization of strength of a functor. While every functor on **Set** comes with a unique strength, not every profunctor on **Set** comes with a strength. However, the realization of a procontainer comes with a canonical strength, as the following shows

$$\begin{aligned}&\mathsf{strength} : \forall\ \{PC\}\ \{X\ Y\ Z\} \to [\![\ PC\ ]\!]\ X\ Y \to [\![\ PC\ ]\!]\ (X \times Z)\ (Y \times Z) \\ &\mathsf{strength}\ (t\ ,\ h) = t\ ,\ \lambda\ \{\ (x\ ,\ z) \to (\mathsf{proj}_1\ (h\ x))\ ,\ \lambda\ p \to (\mathsf{proj}_2\ (h\ x)\ p)\ ,\ z\ \}\end{aligned}$$

which satisfies the axioms of a strong profunctor as postulated by Paré and Román [15]. Moreover, this strength is not only canonical, but also unique for the procontainer realization. An important point here is to see that in the realization of a procontainer, we have that the contravariant argument is used linearly, so there is no choice for the Z in the output for a strength, it must come from the only occurrence provided as input.

Example 1 *The most basic example we can give of a procontainer is the hom-set profunctor on* **Set***, which we can obtain by letting shapes and (both) positions of the procontainer to be unit types:* $\mathsf{Hom} = \top \triangleright (\lambda\ \{\ \mathsf{tt} \to \top\ \}) \triangleright \lambda\ \{\ \mathsf{tt}\ \mathsf{tt} \to \top\ \}$ *where* $\top$ *represents the unit type whose unique inhabitant is* tt*, i.e. the terminal object in* **Set***,* $1 = \{*\}$*. This corresponds to* $1 \longrightarrow 1 \longrightarrow 1$ *as a bundle.*

Example 2 *For any set* N*, the constant profunctor* $P(X,Y) = N$*, it can be encoded as a procontainer* $0 \longrightarrow N \longrightarrow N$*.*

Example 3 *In the context of arrow handlers, Pieters et al. [17] consider profunctors of the form* $P(X,Y) = S \times X \Rightarrow (D \times (N \Rightarrow Y))$ *for sets* S*,* D*, and* N*. One can think of these profunctors as encoding the signature of an operation that has some static input* S *and some dynamic input* D*. They can be expressed as a procontainer* $S \triangleright (\lambda\ \{\ \_ \to D\ \}) \triangleright (\lambda\ \{\ \_\ \_ \to N\ \})$*.*

Example 4 *A similar example is given by a* signature *made of operations* $\mathrm{op} : \delta \rightsquigarrow \gamma$*, where* δ *and* γ *are the arities (sets) of the operation. When we model the signature with a set* Σ *of operations, and function* src *and* dst *mapping to coarities and arities respectively, we can capture it using the procontainer* $\Sigma \triangleright (\lambda\ \{\ op \to \mathsf{src}\ op\ \}) \triangleright (\lambda\ \{\ op\ \_ \to \mathsf{dst}\ op\ \})$*.*

3.1 Procontainer morphisms

To define morphisms between procontainers, we take as a reference point natural transformations between the realizations of them. Given two procontainers $S \triangleright P^+ \triangleright P^-$ and $T \triangleright Q^+ \triangleright Q^-$, a natural transformation from the realization of $S \triangleright P^+ \triangleright P^-$ to the realization of $T \triangleright Q^+ \triangleright Q^-$ is a natural family of morphisms

$$\alpha_{X,Y} : \sum_{s \in S} \left(X \Rightarrow \sum_{p^+ \in P^+(s)} P^-(s,p^+) \Rightarrow Y \right) \longrightarrow \sum_{t \in T} \left(X \Rightarrow \sum_{q^+ \in Q^+(t)} Q^-(t,q^+) \Rightarrow Y \right)$$

This is equivalent to having a family of natural transformations indexed by S

$$\alpha_{s,X,Y} : \left(X \Rightarrow \sum_{p^+ \in P^+(s)} P^-(s,p^+) \Rightarrow Y \right) \longrightarrow \sum_{t \in T} \left(X \Rightarrow \sum_{q^+ \in Q^+(t)} Q^-(t,q^+) \Rightarrow Y \right)$$

These are equivalent to give an element $f(s) \in T$ for each s, together with a family of natural transformations indexed by S

$$\beta_{s,Y} : \left(\sum_{p^+ \in P^+(s)} P^-(s,p^+) \Rightarrow Y \right) \longrightarrow \sum_{q^+ \in Q^+(f(s))} Q^-(t,q^+) \Rightarrow Y$$

But now, we know that a $\beta_{s,Y}$ is equivalent to a natural transformation between the realization of the polynomial functors $P^+(s) \triangleright P^-(s,-)$ and $Q^+(f(s)) \triangleright Q^-(f(s),-)$. Using the characterization of maps between containers, we know that we can express $\beta_{s,Y}$ as a function $f^s : P^+(s) \to Q^+(f(s))$ and a function $f_{p^+} : Q^-(f(s), f(p^+)) \to P^-(s,p^+)$, obtaining that a morphism between procontainers is given by a triple:

- $f : S \to T$
- $f^s : P^+(s) \to Q^+(f(s))$
- $f_{p^+} : Q^-(f(s), f(p^+)) \to P^-(s,p^+)$

In Agda, we can capture this data using a record, which works in a forward direction for shapes and positive positions, and in a backward direction for negative positions:

```
record _⇒_ (PC₁ : ProContainer) (PC₂ : ProContainer) : Set where
  constructor _▷_▷_
  field shmap : Shape PC₁ → Shape PC₂
        posmap⁺ : ∀ s → Position⁺ PC₁ s → Position⁺ PC₂ (shmap s)
        posmap⁻ : ∀ s p⁺ → Position⁻ PC₂ (shmap s) (posmap⁺ s p⁺)
                         → Position⁻ PC₁ s p⁺
```

As in the case of containers, these morphisms are realized as natural transformations:

```
⟨⟨_⟩⟩ : ∀ {PC₁ PC₂} (f : PC₁ ⇒ PC₂) → ∀ {X Y} → ⟦ PC₁ ⟧ X Y → ⟦ PC₂ ⟧ X Y
⟨⟨ shmap ▷ posmap⁺ ▷ posmap⁻ ⟩⟩ (s , f) =
  shmap s , λ x → let f₁ , f₂ = f x in
                  (posmap⁺ s f₁) , λ { p⁻ → f₂ (posmap⁻ s f₁ p⁻) }
```

The realization of the morphisms defined as above respects the canonical strengths we have given for the realization of procontainers. Procontainers form a category which we denote by **ProCont**, where composition is defined as

```
_·_ : ∀ {PC₁ PC₂ PC₃} → (PC₂ ⇒ PC₃) → (PC₁ ⇒ PC₂) → PC₁ ⇒ PC₃
(shmap₁ ▷ posmap⁺₁ ▷ posmap⁻₁) · (shmap₂ ▷ posmap⁺₂ ▷ posmap⁻₂) =
  (λ s → shmap₁ (shmap₂ s)) ▷
  (λ s p⁺ → posmap⁺₁ (shmap₂ s) (posmap⁺₂ s p⁺)) ▷
  (λ s p⁺ p⁻ → posmap⁻₂ s p⁺ (posmap⁻₁ (shmap₂ s) (posmap⁺₂ s p⁺) p⁻))
```

and identities are basically identity functions on each component.

4 Procontainer operations

As discussed above, containers are closed under certain operations: coproducts, products, Day convolution and composition. This is one of the reasons why the class of containers is so successful, as many data-types can be built using these operations. We seek to have a similar closure property in procontainers, where indeed the situation is similar: they are closed under coproducts, products, some forms of convolution, and a tensor corresponding to that of Bénabou.

4.1 Product and coproduct of procontainers

The product of profunctors $P,Q : \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \to \mathbf{Set}$ is given by the pointwise product: $(P \times Q)(X,Y) = P(X,Y) \times Q(X,Y)$. Procontainers are closed under this operation, where the corresponding Agda code is the following:

```
(S ▷ P⁺ ▷ P⁻) × (T ▷ Q⁺ ▷ Q⁻) = (S × T) ▷ (λ { (s , t) → P⁺ s × Q⁺ t })
  ▷ λ { (s , t) (p⁺ , q⁺) → P⁻ s p⁺ ⊎ Q⁻ t q⁺ }
```

Coproduct of profunctors is also defined in a pointwise manner, and procontainers are also closed under this operation, this time taking coproducts of shapes, and keeping account of the branch for positions:

```
(S ▷ P⁺ ▷ P⁻) + (T ▷ Q⁺ ▷ Q⁻) =
  (S ⊎ T) ▷ (λ { (inj₁ s) → P⁺ s ; (inj₂ t) → Q⁺ t })
  ▷ λ { (inj₁ s) s⁺ → P⁻ s s⁺ ; (inj₂ t) t⁺ → Q⁻ t t⁺ }
```

Theorem 1 *The category* **ProCont** *has binary products and coproducts defined by the formulas above.*

Moreover, we can also extend these constructions to the coproduct of a family of procontainers indexed by a set of elements as follows:

```
Σ : ∀ {I : Set} (f : I → ProContainer) → ProContainer
Σ {I} f = (Σ[ i ∈ I ] Shape (f i)) ▷ (λ { (i , s) → Position⁺ (f i) s })
            ▷ λ { (i , s) s⁺ → Position⁻ (f i) s s⁺ }
```

4.2 Tensor of procontainers

Categorically speaking, profunctors are organized in a bicategory consisting of categories, profunctors and natural transformations. The vertical composition is given by the composition of natural transformations, while the horizontal composition for profunctors $P : \mathbb{C} \nrightarrow \mathbb{D}$ and $Q : \mathbb{D} \nrightarrow \mathbb{E}$ is defined using the following coend:

$$(Q \circ P : \mathbb{C} \nrightarrow \mathbb{E})\,(X,Y) = \int^{I \in \mathbb{D}} P(X,I) \times Q(I,Y)$$

This operation is sometimes referred to as Bénabou's tensor, and we write it as $\otimes$ when fixing it a single object (i.e. $\mathbb{C} = \mathbb{D} = \mathbb{E}$). As a monoidal structure, its unit is given by the hom-set profunctor Hom. In the categorical semantics of effects, this tensor is used to justify the point of view of arrows as monoid objects [8, 4]: the sequential composition of arrow computations is given by

$$\_ \ggg \_ : \forall\, \{x\ y\ z\} \to \mathsf{A}\ x\ y \to \mathsf{A}\ y\ z \to \mathsf{A}\ x\ z$$

which can be thought as a natural transformation $A \otimes A \to A$, since families of morphisms $\alpha_{X,Z} : (A \otimes A)(X,Z) \to A(X,Z)$ (natural in X and Z) are classified by the coend as families $\alpha'_{X,Y,Z} : A(X,Y) \times A(Y,Z) \to A(X,Z)$, natural in X, Z and dinatural in Y.

Inspired by this definition, we propose a tensor for procontainers that behaves like Bénabou's tensor:

$$\begin{array}{l}(S \triangleright P^+ \triangleright P^-) \otimes (T \triangleright Q^+ \triangleright Q^-) = \\ \quad (S \times T) \triangleright (\lambda\ \{\ (s\ ,\ t) \to \Sigma[\ s^+ \in P^+\ s\ ]\ (P^-\ s\ s^+ \to Q^+\ t)\ \}) \triangleright \\ \quad \lambda\ \{\ (s\ ,\ t)\ (s^+\ ,f) \to \Sigma[\ s\text{-} \in P^-\ s\ s^+\ ]\ Q^-\ t\ (f\ s\text{-})\ \}\end{array}$$

This operation comes with its corresponding action on morphisms

$$\_ \otimes_m \_ : \forall\, \{PC_1\ PC_2\ PC_3\ PC_4\} \to (PC_1 \Rightarrow PC_2) \to (PC_3 \Rightarrow PC_4) \to (PC_1 \otimes PC_3) \Rightarrow (PC_2 \otimes PC_4)$$

We will not give details, but this tensor is associative. The unit for this version of the Bénabou's tensor is given by the hom-set profunctor Hom. We can write left and right unitality isomorphisms with respect to $\otimes$, thus obtain the following result:

Theorem 2 *The category* **ProCont** *is a monoidal category with Bénabou's tensor* $\otimes$ *defined above and* Hom *as its unit.*

4.3 Convolutions

As shown in Section 2, containers are closed under a convolution operation known as *Day convolution*. This convolution, when applied to F and G, classifies natural families of morphisms

$$m_{A,B} : FA \times GB \longrightarrow H(A \times B)$$

such that they can be expressed as morphisms $F \star G \to H$.

We can consider different monoidal structures in both $\mathbf{Set}^{\mathrm{op}}$ (contravariant argument) and **Set** (covariant argument), and see for which of these we can classify morphisms of the corresponding form.

The convolution we show here instead is the convolution between P and Q that classifies families of morphisms

$$m_{A,B,C,D} : P(A,B) \times Q(C,D) \longrightarrow R(A+C, B+D)$$

natural in A,B,C,D. Procontainers are closed w.r.t. this convolution, and the corresponding result is given by

$$
\begin{aligned}
&(S \triangleright P^+ \triangleright P^-) \mathbin{+\!\star\!+} (T \triangleright Q^+ \triangleright Q^-) = \\
&\quad (S \times T) \triangleright (\lambda \{ (s\ ,\ t) \to (P^+\ s) \uplus Q^+\ t \}) \triangleright \\
&\qquad \lambda \{ (s\ ,\ t)\ (\mathsf{inj}_1\ p^+) \to P^-\ s\ p^+ \ ;\ (s\ ,\ t)\ (\mathsf{inj}_2\ q+) \to Q^-\ t\ q+ \}
\end{aligned}
$$

Families $m_{A,B,C,D} : P(A,B) \times Q(C,D) \longrightarrow R(A \times C, B + D)$ natural in A,B,C,D can also be classified with a convolution, but this case coincides with the product formula for procontainers.

5 Relationship between containers and procontainers

We are now interested in understanding how to convert a profunctor into a functor and vice-versa. Converting a profunctor into a functor is easy: we can fix the first component to a particular set. We will be interested mainly in the case where we fix the first component to the terminal object 1. We can see what is happening for the case of procontainers w.r.t. the realization as profunctors:

$$\llbracket S \triangleright P^+ \triangleright P^- \rrbracket\,(1,Y) \cong \sum_{s \in S} \sum_{p^+ \in P^+(s)} P^-(s,p^+) \Rightarrow Y$$

which can be seen as a container with shapes $\sum_{s \in S} P^+(s)$. In Agda, we express this functor as follows:

$$
\begin{aligned}
&\_^* : \mathsf{ProContainer} \to \mathsf{Container} \\
&(S \triangleright P^+ \triangleright P^-)^* = (\Sigma[\ s \in S\ ]\ P^+\ s) \triangleright \lambda \{ (s\ ,\ p^+) \to P^-\ s\ p^+ \}
\end{aligned}
$$

In terms of bundles, the action of this morphism is to forget the tail and keep only the head:

$$F \xrightarrow{q} E \xrightarrow{p} B \qquad \mapsto \qquad F \xrightarrow{q} E$$

In the opposite direction, given an endofunctor $F : \mathbf{Set} \to \mathbf{Set}$, we can think of two ways of seeing this as a profunctor $\mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \to \mathbf{Set}$: Kleisli and Cayley constructions as presented in the introduction.

5.1 Kleisli

The Kleisli construction takes a functor $F : \mathbf{Set} \to \mathbf{Set}$ to a profunctor $F_* : \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \to \mathbf{Set}$ defined by $F_*(X,Y) = X \Rightarrow FY$. This functor is interesting in many aspects, and it can be seen as a categorification of the assignment turning a function into a relation. In here, we are mostly interested in its usage in the context of computational effects. We can think of it as the formal justification that allows us to transform monads into arrows: if monads are monoids w.r.t. composition, and arrows are monoids w.r.t. Bénabou's tensor, then if there is a monoidal functor between the underlying categories, then we can translate monoids to monoids.

As we did with the transformation from profunctors to functors, we can look at the action of this transformation in the case that F is a container:

$$\llbracket S \triangleright P \rrbracket_*\,(X,Y) \cong X \Rightarrow \sum_{s \in S} Ps \Rightarrow Y$$

In our setting, we can transform the right term into a procontainer: we choose $S = 1$, $P^+(*) = S$ and $P^-(s) = P(s)$. Thus, we can make a functor **Cont** $\to$ **ProCont** that sends a container to this corresponding procontainer:

$$\__* : \text{Container} \to \text{ProContainer}$$
$$(S \triangleright P)_* = \top \triangleright (\lambda \{ \text{tt} \to S \}) \triangleright \lambda \{ \text{tt}\ s \to P\ s \}$$

In terms of bundles, this functor adds a tail using the terminal map:

$$E \xrightarrow{p} B \qquad \mapsto \qquad E \xrightarrow{p} B \xrightarrow{!_B} 1$$

This functor works as a right adjoint to the functor $\_^*$:

Theorem 3 *There is an adjunction* $\_^* \dashv \__*$.

The functor $\__*$ is a *strong monoidal functor* with respect to the structures $(\circ, \mathrm{Id})$ in **Cont** and $(\otimes, \mathrm{Hom})$ in **ProCont**. It means that we have isomorphisms:

$$\phi_0 : \mathrm{Hom} \cong \mathrm{Id}_* \qquad \phi_{C,D} : (C_*) \otimes (D_*) \cong (C \circ D)_* \text{ natural in } C \text{ and } D$$

Using the monoidality of the functor, we can map monoids to monoids. Given a monoid (T, μ, η) in $(\mathbf{Cont}, \circ, \mathrm{Id})$, which can be realized as a monad, we can obtain a new monoid $(T_*, \mu_* \cdot \phi_{T,T}, \eta_* \cdot \phi_0)$ in $(\mathbf{ProCont}, \otimes, \mathrm{Hom})$ which can be realized as an arrow, and this is the translation Kleisli-monad presented in the introduction.

5.2 Cayley

We are now interested in visiting the so-called *Cayley functor*. This was originally introduced by Pastro and Street in the context of enriched strong profunctors, under the name of *Tambara modules* [16].

The Cayley construction takes a functor $F : \mathbf{Set} \to \mathbf{Set}$ to a profunctor $F_! : \mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \to \mathbf{Set}$ defined as $F_!(X,Y) = F(X \Rightarrow Y)$. We can see its action on the realization of a container $S \triangleright P$,

$$[\![S \triangleright P]\!]_!(X,Y) \cong \sum_{s \in S} X \Rightarrow [Ps \Rightarrow Y]$$

which we can be understood as the realization of a procontainer with trivial P^+.

$$\__! : \text{Container} \to \text{ProContainer}$$
$$(S \triangleright P)_! = S \triangleright (\lambda \_ \to \top) \triangleright \lambda \{ s\ \text{tt} \to P\ s \}$$

Now, when containers are seen as bundles, this functor adds a tail using an identity:

$$E \xrightarrow{p} B \qquad \mapsto \qquad E \xrightarrow{p} B \xrightarrow{\mathrm{id}} B$$

In this case, this functor works as a left adjoint to $\_^*$:

Theorem 4 *There is an adjunction* $\__! \dashv \_^*$.

As in the general case shown by Pastro and Street, this functor is strong monoidal from Day convolution in containers to Bénabou's tensor in procontainers. It means, again, that we have a way to map monoids from $(\mathbf{Cont}, \star, \mathrm{Id})$ to monoids in $(\mathbf{ProCont}, \otimes, \mathrm{Hom})$, which reproduces the transformation Cayley-idiom seen in the introduction.

6 Relationship between Dirichlet functors and procontainers

We can take some of the observations in the previous section, but replacing containers with Dirichlet functors. We obtain a Dirichlet functor from a procontainer by fixing the second argument to the terminal object of **Set**, i.e. 1:

$$[\![S \triangleright P^+ \triangleright P^-]\!]\,(X,1) \cong \sum_{s \in S} \left(X \Rightarrow P^+(s)\right)$$

which is the realization of the Dirichlet functor $S \triangleright P^+$. In code, we write this transformation as:

```
_¹ : ProContainer → Dirichlet
(S ▷ P⁺ ▷ _) ¹ = S ▷ P⁺
```

When seen in term of bundles, what we are doing is only keeping the tail:

$$F \xrightarrow{q} E \xrightarrow{p} B \qquad \mapsto \qquad E \xrightarrow{p} B$$

We can see that this is less satisfying than the analogous functor to containers, as here we are losing more information: B information is encoded in E so we can throw the tail away without much remorse, but it is not the case when keeping only the tail, as the information of F is lost. In any case, we can still find left and right adjoints to these functors following the same recipes as in containers:

$$E \xrightarrow{\mathrm{id}} E \xrightarrow{p} B \quad \overset{-_1}{\hookleftarrow} \quad E \xrightarrow{p} B \quad \overset{-_0}{\mapsto} \quad 0 \xrightarrow{\mathrm{i}_E} E \xrightarrow{p} B$$

which we write in Agda as:

```
_₁ : Dirichlet → ProContainer
(S ▷ P) ₁ = S ▷ P ▷ λ s s⁺ → ⊤
```

```
_₀ : Dirichlet → ProContainer
(S ▷ P) ₀ = S ▷ P ▷ λ _ _ → ⊥
```

where $\bot$ represents the empty type, i.e. initial object in **Set**, $0 = \emptyset$.

Theorem 5 *There are adjunctions* `_₁ ⊣ _₀` *and* `_¹ ⊣ _₀`.

When we look at the action of these in terms of functors, we find that

$$(F_1)(X,Y) = F(X) \times (X \Rightarrow Y) \qquad\qquad (F_0)(X,Y) = F(X)$$

7 Generalization, limitations and related work

In this article we have considered only profunctors of the form $\mathbf{Set}^{\mathrm{op}} \times \mathbf{Set} \to \mathbf{Set}$, or "*endoprofunctors on* **Set**". We now discuss briefly possible generalizations of procontainers to more general categories.

When we change **Set**, we might have two possible generalizations, either we generate functors of the form $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{C}$, or profunctors $\mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbf{Set}$. For this last version, we can consider the construction of procontainers as follows. Given a functor $F : \mathbb{C} \to \mathbb{C}$, we can construct an endoprofunctor $F_*(X,Y) = \mathrm{Hom}(X, F(Y))$ which the external version for the Kleisli construction we introduced in Section 5. Such profunctors are called representable. We can extend this construction with coproducts as follows: given

a family of functors $\{F^s : \mathbb{C} \to \mathbb{C}\}_{s \in S}$, we can construct a new profunctor that is the coproduct of the corresponding representable profunctors:

$$F_*(X,Y) = \sum_{s \in S} F^s_*(X,Y) = \sum_{s \in S} \mathrm{Hom}(X, F^s(Y))$$

Under this view, a procontainer on $\mathbb{C}$ can be thought of as a family $\left\{P^s_+ \triangleright P^s_-\right\}_{s \in S}$ of containers on $\mathbb{C}$, which is realized as the profunctor corresponding to the coproduct of representable profunctors coming from the representation of the containers in the family, i.e.

$$\left[\!\left[\left\{P^s_+ \triangleright P^s_-\right\}_{s \in S}\right]\!\right](X,Y) = \sum_{s \in S} \mathrm{Hom}\left(X, \sum_{p^+ \in P^s_+} \left(P^s_-(p^+) \Rightarrow Y\right)\right)$$

Notice that the outer coproduct is a coproduct in **Set**, while the inner coproduct is a coproduct in $\mathbb{C}$, and at the same time, Hom represents the external hom of $\mathbb{C}$, while $- \Rightarrow -$ represents the internal hom of $\mathbb{C}$.

As for limitations, we have left out the important case where a container is a comonad. These containers are the *directed containers* of Ahman and Uustalu [2]. Sadly, we cannot treat them naturally as we did with monads. To see this, notice that comonads are embedded as profunctors not using $-^*$ but instead using its "dual" $F^{/}(X,Y) = FX \Rightarrow Y$. Calculating the action of this functor on a container $S \triangleright P$, we obtain:

$$\left(\sum_{s \in S} (P(s) \Rightarrow X)\right) \Rightarrow Y \cong \prod_{s \in S} (P(s) \Rightarrow X) \Rightarrow Y$$

which cannot be understood as a procontainer in principle: it could have multiple positions for X (depending on $P(s)$), which break our ability to write a proper strength. More generally, the strength for the profunctor obtained by a comonad uses the counit of the comonad, which in principle cannot be given naturally. This problem stops us from being able to treat interesting arrows which come from a distributive law [8, 21]. One might try alternatives by having a different notion of realization for procontainer, such as a *product of corepresentable profunctors* instead, although, it is not obvious how to handle strength in that case.

8 Conclusion

Guided by interfaces for computational effects, we have introduced a notion of *procontainer*. We have shown that this notion is closed under interesting operators, and moreover, it has interesting connections to containers which reflect connections to the treatment computational effects in functional programming and semantics of programming languages.

It would be interesting to find more categorical characterizations of procontainers, such as those preserving certain forms of (co)limits. We leave this for further investigation, as well as other categorical properties that the category of containers has, such as being Cartesian closed [3], or being able to interpret a model of type theory [6].

Acknowledgments The author is grateful to Tarmo Uustalu for useful discussions and comments on an early version of this paper. This research was supported by the Estonian Research Council grant no. PSG749 and the Icelandic Research Fund project grant 228684-051.

References

[1] Michael Abbott, Thorsten Altenkirch & Neil Ghani (2003): *Categories of Containers*. In: *Proceedings of the 6th International Conference on Foundations of Software Science and Computation Structures and Joint European Conference on Theory and Practice of Software*, FOSSACS'03/ETAPS'03, Springer-Verlag, Berlin, Heidelberg, pp. 23–38, doi:10.1007/3-540-36576-1_2.

[2] Danel Ahman & Tarmo Uustalu (2016): *Directed Containers as Categories*. In: *Proceedings 6th Workshop on Mathematically Structured Functional Programming*, *EPTCS* 207, pp. 89–98, doi:10.4204/EPTCS.207.5.

[3] Thorsten Altenkirch, Paul Levy & Sam Staton (2010): *Higher-Order Containers*. In: *Proceedings of the Programs, Proofs, Process and 6th International Conference on Computability in Europe*, CiE'10, Springer-Verlag, Berlin, Heidelberg, p. 11–20, doi:10.1007/978-3-642-13962-8_2.

[4] Kazuyuki Asada (2010): *Arrows Are Strong Monads*. In Venanzio Capretta & James Chapman, editors: *Proceedings of the Third ACM SIGPLAN Workshop on Mathematically Structured Functional Programming*, MSFP '10, ACM, pp. 33–42, doi:10.1145/1863597.1863607.

[5] Nicola Gambino & Joachim Kock (2013): *Polynomial functors and polynomial monads*. *Mathematical Proceedings of the Cambridge Philosophical Society* 154(1), p. 153–192, doi:10.1017/S0305004112000394.

[6] Tamara von Glehn (2015): *Polynomials and models of type theory*. Ph.D. thesis, doi:10.17863/CAM.16245.

[7] John Hughes (2000): *Generalising Monads to Arrows*. *Science of Computer Programming* 37(1-3), pp. 67–111, doi:10.1016/S0167-6423(99)00023-4.

[8] Bart Jacobs, Chris Heunen & Ichiro Hasuo (2009): *Categorical semantics for arrows*. *Journal of Functional Programming* 19(3-4), pp. 403–438, doi:10.1017/S0956796809007308.

[9] Joachim Kock (2010): *Polynomial Functors and Trees*. *International Mathematics Research Notices* 2011(3), pp. 609–673, doi:10.1093/imrn/rnq068.

[10] Sam Lindley, Philip Wadler & Jeremy Yallop (2011): *Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous*. *Electronic Notes on Theoretical Computer Science* 229(5), pp. 97–117. Available at `http://dx.doi.org/10.1016/j.entcs.2011.02.018`.

[11] Connor McBride & Ross Paterson (2008): *Applicative programming with effects*. *Journal of Functional Programming* 18(01), pp. 1–13, doi:10.1017/S0956796807006326.

[12] Eugenio Moggi (1989): *Computational lambda-calculus and monads*. In: *Fourth Annual Symposium on Logic in Computer Science*, pp. 14–23, doi:10.1109/LICS.1989.39155.

[13] Eugenio Moggi (1991): *Notions of computation and monads*. *Inf. Comput.* 93(1), pp. 55 – 92, doi:10.1016/0890-5401(91)90052-4.

[14] David Jaz Myers & David I. Spivak (2020): *Dirichlet Functors are Contravariant Polynomial Functors*, doi:10.48550/ARXIV.2004.04183.

[15] Robert Paré & Leopoldo Román (1998): *Dinatural numbers*. *Journal of Pure and Applied Algebra* 128(1), pp. 33–92, doi:https://doi.org/10.1016/S0022-4049(97)00036-4.

[16] Craig Pastro & Ross Street (2008): *Doubles for monoidal categories*. *Theory and Applications of Categories* 21, pp. 61–75.

[17] Ruben P. Pieters, Exequiel Rivas & Tom Schrijvers (2020): *Generalized monoidal effects and handlers*. *Journal of Functional Programming* 30, p. e23, doi:10.1017/S0956796820000106.

[18] Exequiel Rivas (2018): *Relating Idioms, Arrows and Monads from Monoidal Adjunctions*. In: *Proceedings of the 7th Workshop on Mathematically Structured Functional Programming*, doi:10.4204/EPTCS.275.3.

[19] David I. Spivak (2020): *Poly: An abundant categorical setting for mode-dependent dynamics*, doi:10.48550/ARXIV.2005.01894.

[20] David I. Spivak (2022): *A reference for categorical structures on* **Poly**, doi:10.48550/ARXIV.2202.00534.

[21] Tarmo Uustalu & Varmo Vene (2005): *Signals and Comonads*. *Journal of Universal Computer Science* 11(7), pp. 1310–1326, doi:10.3217/jucs-011-07-1311.

[22] Philip Wadler (1992): *The Essence of Functional Programming*. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, Association for Computing Machinery, New York, NY, USA, p. 1–14, doi:10.1145/143165.143169.