Freer Arrows and Why You Need Them

GRANT VANDOMELEN, Portland State University, USA
YAO LI, Portland State University, USA

Freer monads are a useful structure commonly used in various domains due to its expressiveness. However, a known issue with freer monads is that they are not amenable to static analysis. This paper explores freer arrows, a structure that is relatively expressive and amenable to static analysis. We propose several variants of freer arrows, including basic freer arrows and bridged freer arrows. We define an equivalence relation for freer arrows that compares their semantical parts semantically and their syntactic parts syntactically. Finally, we conduct a few case studies to demonstrate the usefulness of freer arrows.

ACM Reference Format:
Grant VanDomelen and Yao Li. 2025. Freer Arrows and Why You Need Them. *Proc. ACM Program. Lang.* 9, OOPSLA (September 2025), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

We love monads [Moggi 1991; Wadler 1992]. We use them all the time. Why not? They are general. They are abstract. They are expressive. They allow us to do diverse things under the same interface.

For this reason, it should be no surprise that structures like freer monads [Kiselyov and Ishii 2015] and their variants [Capretta 2005; Dylus et al. 2019; McBride 2015; Piróg and Gibbons 2014; Swamy et al. 2020; Xia et al. 2020] have been used in various domains including choreographic programming [Shen et al. 2023], concurrency [Marlow et al. 2014], algebraic effects [Dev et al. 2024; Maguire 2025; Wu et al. 2025], specifications [Letan et al. 2018; Ye et al. 2022; Zhang et al. 2021], embeddings [Chlipala 2021; Christiansen et al. 2019; Korkut et al. 2025], testing [Li et al. 2021], information-flow analysis [Silver et al. 2023; Silver and Zdancewic 2021], *etc.*

However, the more expressive an interface is, the less we know about it.

In particular, the dynamic nature of monads make them impossible to be statically analyzed [Capriotti and Kaposi 2014; Li and Weirich 2022; Mokhov et al. 2019, 2020]. For example, it is impossible to define a function that counts the number of effect invocations in a freer monad without interpreting it, *even if that number does not depend on results of effect invocations.*

In this paper, we explore a different structure that offers an alternative trade-off between expressiveness and static analyzeability: *freer arrows*. Arrows are an abstraction initially proposed by Hughes [2000] as a generalization of monads. Later, Lindley et al. [2008] discover that arrows sit between applicative functors and monads in terms of expressiveness. Although free arrows have been studied by literatures like Rivas and Jaskelioff [2017], *freer* arrows have received little attention. Our paper fills this gap.

We make the following contributions:

- We define basic freer arrows, including *freer pre-arrows*, *freer arrows*, and *freer choice arrows* (Section 3).
- We propose a novel structure called *bridged freer arrows*, a partially-defunctionalized variant of freer arrows that contains less "administrative" code and boilerplates compared with freer arrows (Section 4).

Authors' addresses: Grant VanDomelen, gsv@pdx.edu, Computer Science, Portland State University, Portland, OR, USA; Yao Li, liyao@pdx.edu, Computer Science, Portland State University, Portland, OR, USA.

- We propose a novel equivalence relation for freer arrows and freer choice arrows that are based on *characteristic functions* and *similarity* between two freer arrows that equate two freer arrows even when they differ in inner existential types. We show that freer arrows and freer choice arrows are lawful arrows based on the equivalence relation (Section 5).
- We present two case studies with freer arrows: a smaller one on states and simple key-value stores, and a larger one based on choreographic programming framework HasChor [Shen et al. 2023] (Section 6–7).

In addition, we discuss the background in Section 2 and related work in Section 8. We conclude this paper and discuss future work in Section 9.

2 BACKGROUND

2.1 Arrows and Profunctors

Arrows were proposed by Hughes [2000] as a generalization of monads. It was later that people realized that arrows are *profunctors* [Asada 2010; Asada and Hasuo 2010; Jacobs et al. 2009]. For this reason, libraries on arrows and profunctors are not organized together in Haskell. For consistency of presentation, we instead present the definitions of arrows and profunctors together in this paper. However, all the ideas and techniques discussed in this paper apply to using arrows from Haskell's `base` library or the `profunctor` library[1] as well. In fact, the code in our supplementary materials is mostly based on `base` and `profunctor` libraries.

We show the definition of profunctors and arrows in Fig. 1. A `Profunctor` is parameterized by two arguments (line 1). It is *contravariant* on its first argument and *covariant* on its second argument, as indicated by its `dimap` method (line 2). It additionally includes a method `lmap` specifically for the contravariant argument and a method `rmap` for the covariant argument, but we omit them here as they can be defined using `dimap`. Function arrows are a classic instance of `Profunctor`, as its argument is contravariant and its return value is covariant.

A `StrongProfunctor` is equipped with one additional method `first'` that "transform" a profunctor `p a b` to `p (a, c) (b, c)` for arbitrary `c` (lines 4–5). A `ChoiceProfunctor` is similar to a `StrongProfunctor`, but its `left'` method works on sum types instead of product types (lines 7–8). There is also a `second'` method for `StrongProfunctors` and a `right'` for `ChoiceProfunctors` that can be defined in terms of `first'` and `left'`, so we omit them here. Function arrows are both strong profunctors and choice profunctors.

Arrows start at `Category`. A `Category` has two methods: **`id`** for identity category (line 11) and `(.)` for category compositions (lines 12–13). A `PreArrow` is a `Category` with an additional `arr` method that takes a function to produce a category (lines 15–16). `Arrows` add a `first` method to a `PreArrow` that shares the same type as the `first'` method of `StrongProfunctors`. Similarly, `ChoiceArrows` add a `left` methods to a `PreArrow` that shares the same type as the `left'` method of `ChoiceProfunctors`. Function arrows are arrows and choice arrows.

Lines 24–34 show that pre-arrows are profunctors, arrows are strong profunctors, and choice arrows are choice profunctors.

Lines 36–39 define a commonly used operator `(>>>)` for categories, which is essentially flipping the arguments of `(.)`.

Profunctors and arrows are expected to follow certain laws. We show these laws in Appendix A.

[1] https://github.com/ekmett/profunctors/

```haskell
class Profunctor (p :: Type -> Type -> Type) where
  dimap :: (a -> b) -> (c -> d) -> p b c -> p a d

class Profunctor p => StrongProfunctor p where
  first' :: p a b -> p (a, c) (b, c)

class Profunctor p => ChoiceProfunctor p where
  left' :: p a b -> p (Either a c) (Either b c)

class Category (cat :: k -> k -> Type) where
  id  :: forall (a :: k). cat a a
  (.) :: forall (b :: k) (c :: k) (a :: k).
         cat b c -> cat a b -> cat a c

class Category a => PreArrow a where
  arr :: (b -> c) -> a b c

class PreArrow a => Arrow a where
  first :: a b c -> a (b, d) (c, d)

class PreArrow a => ChoiceArrow a where
  left :: a b c -> a (Either b d) (Either c d)

-- PreArrows are Profunctors
instance PreArrow a => Profunctor a where
  dimap f g a = arr f >>> a >>> arr g

-- Arrows are StrongProfunctors
instance Arrow a => StrongProfunctor a where
  first' = first

-- ChoiceArrows are ChoiceProfunctors
instance ChoiceArrow a => ChoiceProfunctor a where
  left' = left

-- A commonly used operator for categories
(>>>) :: forall k (a :: k) (b :: k) (c :: k) (cat :: k -> k -> Type).
  Category cat => cat a b -> cat b c -> cat a c
(>>>) = flip (.)
```

Fig. 1. Definitions of `Profunctor` and `Strong` typeclasses. For conciseness, we only include typeclass methods that are necessary for a minimal implementation. In Haskell's `profunctor` library, `StrongProfunctor` is simply called `Strong` and `ChoiceProfunctor` is called `Choice`. We use the full names in this paper to avoid potential confusions.

2.2 The Expressiveness of Arrows

In Lindley et al. [2008], the authors discover that arrows sit between applicative functors and monads in terms of expressiveness. They give an example based on the following state monad interface:

```
get :: unit -> M Int
put :: Int -> M unit
```

We use `M` to represent a monad here.

In their paper, Lindley et al. show that we can define the following two functions using the monad interface:

```
freshName :: M Name
freshName = get () >>= (\s -> put (s + 1) >>= (\u -> return (makeName s)))

ifZero :: (M A, M A) -> M A
ifZero k = get () >>= (\s -> if s == 0 then fst k else snd k)
```

If we instead use an arrow interface here (all monads can be converted to arrows using a Kleisli construction shown by Hughes [2000]), we can only define `freshName`, but not `ifZero`. If we instead use an applicative functor interface, we would not be able to define either `freshName` or `ifZero`. This shows that arrows are between applicative functors and monads in terms of expressiveness power.

Furthermore, a *static* arrow `arr` that has an isomorphism between `arr () (i -> o)` and `arr i o` is known to correspond to applicative functors. An `ArrowApply arr` that has an additional method `app :: arr (arr i o, i) o` corresponds to a monad. Later, Mokhov et al. [2019] further show that a choice arrow corresponds to selective functors.

2.3 Free Profunctors and Free Arrows

In their paper, Rivas and Jaskelioff [2017] define the following version of free arrows:

```
data Free a x y where
  Hom :: (x -> y) -> Free a x y
  Comp :: a x z -> Free a z y -> Free a x y
```

They further show that `Free` is a profunctor and a pre-arrow if `a` is a profunctor, and `Free` is a strong profunctor and an arrow if `a` is a strong profunctor, given by the following definitions:

```
instance Profunctor a => Profunctor (Free a) where
  dimap f g (Hom h)    = Hom (g . h . f)
  dimap f g (Comp x y) = Comp (lmap f x) (rmap g y)

instance Profunctor a => PreArrow (Free a) where
  arr f = Hom f
  c . (Hom f)    = lmap f c
  c . (Comp x y) = Comp x (c . y)

instance StrongProfunctor a => Arrow (Free a) where
  first (Hom f)    = Hom (first f)
  first (Comp x y) = Comp (first' x) (first' y)

instance StrongProfunctor a => StrongProfunctor (Free a) where
  first' = first
```

However, free arrows are somewhat unsatisfying as they require their first parameter to be a strong profunctor. We want to make free arrows *freer* by lifting this restriction. This is important because we want to use freer arrows with generic algebraic datatypes (GADTs) like the following:

```
-- |- A GADT for stateful effect.
data StateEff :: Type -> Type -> Type -> Type where
  Get :: StateEff s a s
  Put :: StateEff s s s
```

Such a GADT is not even a profunctor because one cannot define the `dimap` method on it.

This is where our work comes in.

3 BASIC FREER ARROWS

We present two variants of freer arrows: *basic* freer arrows and *bridged* freer arrows. Basic freer arrows (or just freer arrows for short)[2] have a simple structure but their definitions requires a fair amount of "administrative" code to make types align. Therefore, we also proposed a bridged version of freer arrows that avoid these issues. We first present freer arrows in this section. We will introduce bridged freer arrows in Section 4.

3.1 Freer Pre-Arrows

We show the definition of freer pre-arrows in Fig. 2. A freer pre-arrow is parameterized by an effect datatype `e :: Type -> Type -> Type`, an "input" datatype `x :: Type`, and an "output" datatype `y :: Type` (line 1). There are only two constructors in a freer pre-arrow. The first constructor, `Hom`, simply wraps a function of type `x -> y` inside it (line 2). The second constructor, `Comp`, is the key for embedding effects and composing freer arrows (lines 3–4). There are two existential types in the `Comp` constructor, namely `a` and `b` (*i.e.,* they don't appear in the type of `FreerArrow`). The `Comp` constructor takes three arguments. First, there is a function argument `x -> a` that does some pure computation that transform an `x` to type `a` (line 3). The value of type `a` is then passed to the effect `e a b` that outputs a value of type `b` (line 3). Finally, there is another `FreerArrow` that takes the value of type `b` and returns `y` (line 4).

The definition of `FreerPreArrow` can be obtained by inlining free profunctors in free arrows (Section 2.3), similar to how Kiselyov and Ishii [2015] derive freer monads. The gives you:

```
data FreerArrowB e x y where
  Hom :: (x -> y) -> FreerArrowB e x y
  Comp :: (x -> a) ->  (b -> z) -> e a b ->
          FreerArrowB e z y -> FreerArrowB e x y
```

From this definition, we take one additional step by fusing the second function argument `b -> z` to the function argument of the "inner" `FreerArrow`, which would have type `z -> c`, where `c` is a new existential type.

We also show that `FreerPreArrows` are instances of `Profunctor`, `Category`, and `PreArrow` in lines 6–24 of Fig. 2. When defining `Profunctor`'s `dimap` method, we pattern match on the `FreerPreArrow`. In the case of `Hom h`, we use function composition `(.)` to compose `h` with both the "contravariance function" and "covariance function" (line 8). In the case of `Comp f' x y`, we compose the "contravariance function" with `f'` and pass the "covariance function" to the recursive call (line 9). Typically, `dimap` is all we need to implement a `Profunctor` instance, as other functions like `lmap` and `rmap` can be implemented using `dimap`. However, we implement `lmap` separately for `FreerPreArrow`

[2]In this paper, we sometimes use freer arrows to refer to all variants of freer arrows, we sometimes use the term to refer to the specific definition `FreerArrow`. The context should be clear to distinguish the use of this term.

```
1  data FreerPreArrow e x y where
2    Hom   :: (x -> y) -> FreerPreArrow e x y
3    Comp  :: (x -> a) -> e a b ->
4             FreerPreArrow e b y -> FreerPreArrow e x y
5
6  -- |- Freer pre-arrows are profunctors.
7  instance Profunctor (FreerPreArrow e) where
8    dimap f g (Hom h) = Hom (g . h . f)
9    dimap f g (Comp f' x y) = Comp (f' . f) x (dimap id g y)
10
11   -- lmap can be implemented more efficiently without recursion
12   lmap f (Hom h) = Hom (h . f)
13   lmap f (Comp f' x y) = Comp (f' . f) x y
14
15 -- |- Freer pre-arrows are categories.
16 instance Category (FreerPreArrow e) where
17   id = Hom id
18
19   f . (Hom g) = lmap g f
20   f . (Comp f' x y) = Comp f' x (f . y)
21
22 -- |- Freer pre-arrows are pre-arrows.
23 instance PreArrow (FreerPreArrow e) where
24   arr = Hom
25
26 -- |- The type for effect handlers.
27 type x :-> y = forall a b. x a b -> y a b
28
29 -- |- Freer pre-arrows can be interpreted into any pre-arrows, as long as we
30 -- can provide an effect handler.
31 interp :: (Profunctor arr, PreArrow arr) =>
32   (e :-> arr) -> FreerPreArrow e x y -> arr x y
33 interp _       (Hom f)       = arr f
34 interp handler (Comp f x y)  = lmap f (handler x) >>> interp handler y
```

Fig. 2. Key definitions of freer pre-arrows (`FreerPreArrow`) in Haskell.

because it can be implemented efficiently without any recursive calls (lines 11–13). This is crucial for performance of freer arrows, as a `FreerPreArrow` can be quite long. Showing that `FreerPreArrows` are an instance of `Category` (lines 15–20) and `PreArrow` (line 22–24) is straightforward by making use of the fact that functions are categories and profunctors.

Finally, we can interpret a `FreerPreArrow` to any pre-arrow if we provide an "effect handler" (lines 26–34). An effect handler has type `e :-> arr` where `e` is an effect type and `arr` is a pre-arrow. We use the type operator `x :-> y` to represent "transformations" from `x a b` to `y a b` for any `a` and `b` (lines 26–27).[3] When interpreting a `FreerPreArrow`, we do a case analysis on the freer pre-arrow.

[3]We borrow this notation from the `profunctors` library by Edward Kmett: https://hackage.haskell.org/package/profunctors-5.6.2

```
1  data FreerArrow e x y where
2    Hom :: (x -> y) -> FreerArrow e x y
3    Comp :: (x -> (a, c)) -> e a b ->
4            FreerArrow e (b, c) y -> FreerArrow e x y
5
6  -- |- Freer arrows are strong profunctors.
7  instance StrongProfunctor (FreerArrow e) where
8    first' (Hom f) = Hom $ first f
9    first' (Comp f a b) =
10     Comp (first f >>> assoc)
11          a (lmap unassoc (first' b))
12
13 -- |- Freer arrows are arrows.
14 instance Arrow (FreerArrow e) where
15   first = first'
16
17 -- |- Freer arrows can be interpreted into any arrows, as long as we can provide
18 -- an effect handler.
19 interp :: (Profunctor arr, Arrow arr) =>
20   (e :-> arr) -> FreerArrow e x y -> arr x y
21 interp _       (Hom f) = arr f
22 interp handler (Comp f x y) = lmap f (first (handler x)) >>>
23                               interp handler y
24
25 -- Helper functions. Definitions omitted.
26 assoc :: ((a,b),c) -> (a,(b,c))
27 unassoc :: (a,(b,c)) -> ((a,b),c)
```

Fig. 3. Key definitions of freer arrows (`FreerArrow`) in Haskell.

In the case of `Hom f`, we simply apply the `arr` method of the pre-arrow `arr` to `f` (line 33). In the case of `Comp f x y`, we use our handler to handle `x`, apply the `lmap` method of the pre-arrow `arr` to both `f` and `handler x`, and compose it with the recursive interpretation of `y` (line 34).

3.2 Freer arrows

To define freer arrows, we need to enhance `FreerPreArrow` with products to enable defining `first`. In fact, we only need to modify the `Comp` constructor. We show the definition of freer arrows in Fig. 3.

Compared with `FreerPreArrow`, the new `Comp` constructor contains three existential types: `a`, `b`, and `c` (lines 3–4). The return type of the function argument is changed to `(a, c)` (from `a`, line 3). The input type of the inner freer arrow is changed to `(b, c)` (from `b`, line 4). The effect type `e a b` remains unchanged. This means that the value of type `c` is simply "passed along" to the next `FreerArrow` without being processed by `e`—this may seem redundant but it's crucial for implementing the `first` method (also known as the *strength* of arrows).

The definition of `FreerArrow` can be obtained by inlining free *strong* profunctors in free arrows and fusing the "covariance function" of a `Comp` constructor with the "contravariance function" of the inner freer arrow, similar to what we did with `FreerPreArrow`.

`FreerArrows` are instances of `Profunctor`, `Category`, and `PreArrow`. These definitions are the same as those of `FreerPreArrow`, so we omit them here. Instead, we show the implementation of `StrongProfunctor` and `Arrow` in lines 6–15 on Fig. 3.

The implementation of `StrongProfunctor` is more tricky. In the case of `Hom f`, we apply the `first` method of the function arrow to `f` (line 8). In the case of `Comp f a b`, we need a few extra steps to take care of types. First, we apply the `first` method of the function arrow to `f`, which gives us a product type in the form of `((_, _), _)`. To match the type with that of the effect `a`, we call `assoc` to convert the product to the form of `(_, (_, _))` (line 10). Finally, we call `unassoc`, the inverse of `assoc` on the recursive evaluation `first' b` (line 11). Here, function calls to `assoc` and `unassoc` are only to align types—they do not pose any computational significance, but we have to carry them around when using `first'` method on `FreerArrows`.

Once we show that `FreerArrow` is a strong profunctor, it is trivial to show that it is also an arrow (lines 14–15).

Finally, if we are provided with an effect handler, we can interpret a freer arrow to any arrows using the `interp` function (lines 17–23). Compared with the `interp` function of `FreerPreArrow`, we need to call `first` on `handler x`, due to the type of `f` (line 22). This is inevitable even for the segments of a freer arrow that we do not use `first`. We talk about how to eliminate the administrative code like `assoc` and `unassoc` and how to remove unnecessary calls to `first` in Section 4.

Showing that a `FreerArrow` is an instance of `Arrow` is not enough for showing that it is indeed an arrow. We also need to show that the instance satisfies arrow laws (Appendix A). However, it turns out that even though `FreerArrow` satisfies the arrow laws, finding the right equivalence relation for `FreerArrow` is challenging, due to the flexibility of arrows. We will discuss the challenges, the equivalence relation, and formal proof of arrow laws in Section 5.

We can statically count the number of events in a freer arrow using the following simple function:

```
count :: FreerArrow e x y -> Int
count (Hom _) = 0
count (Comp _ _ y) = 1 + count y
```

This will always give an exact count, since the sequence of events is known completely statically and is not affected by the data inside the arrow.

3.3 Freer Choice Arrows

We also define freer choice arrows for use cases where conditionals are required. We show the key definitions of freer choice arrows in Fig. 4.

The `Hom` constructor of `FreerChoiceArrow` (line 2) is the same as that of `FreerArrow`. However, the `Comp` constructor adds one additional existential variable `w` (lines 3–6). The function argument can either return a product of type `(a, c)`, just like in `FreerArrow`, or return a value of type `w` (line 3). In the first case, the value of type `a` will be passed to the effect of type `e a b` while the value of type `c` is passed along (line 4). In the second case, the value of type `w` will be passed along without invoking the effect at all. Finally, the inner `FreerChoiceArrow` needs to handle the new input type `Either (b, c) w` (line 5).

We also show the definitions of `first'` and `left'` methods in lines 8–16 of Fig. 4. In the `Hom` cases, we simply use the `first` or `left` methods of function choice arrows (line 9 and line 14).

The `Comp` cases are more complex. For the `StrongProfunctor` instance, we first apply the `first` method of function arrows to `f`, which gives us a type of the form `((Either (_, _) _), _)`. We use the `distr` function defined on line 27 to convert the type to a form of `Either ((_, _), _) (_, _)`.[4] After that, we apply the `left` method of function choice arrows to `assoc` to convert the type from the

[4]This corresponds to the distributivity of multiplication over additions, hence the function name `distr`.

```
1  data FreerChoiceArrow e x y where
2    Hom :: (x -> y) -> FreerChoiceArrow e x y
3    Comp :: (x -> Either (a, c) w) ->
4            e a b ->
5            FreerChoiceArrow e (Either (b, c) w) y ->
6            FreerChoiceArrow e x y
7
8  instance StrongProfunctor (FreerChoiceArrow e) where
9    first' (Hom f) = Hom $ first f
10   first' (Comp f a b) = Comp (first f >>> distr >>> left assoc)
11                             a (lmap (left unassoc >>> undistr) (first' b))
12
13 instance ChoiceProfunctor (FreerChoiceArrow e) where
14   left' (Hom f) = Hom $ left f
15   left' (Comp f a b) = Comp (left f >>> assocsum)
16                             a (lmap unassocsum (left' b))
17
18 -- |- Freer choice arrows can be interpreted into any choice arrows, as long as
19 -- we can provide an effect handler.
20 interp :: (Profunctor arr, ArrowChoice arr) =>
21   (e :-> arr) -> FreerChoiceArrow e x y -> arr x y
22 interp _       (Hom f) = arr f
23 interp handler (Comp f x y) = lmap f (left (first (handler x))) >>>
24                                interp handler y
25
26 -- Helpful functions. Definitions omitted.
27 distr :: (Either (a, b) c, d) -> Either ((a, b), d) (c, d)
28 undistr :: Either ((a, b), d) (c, d) -> (Either (a, b) c, d)
29 assocsum :: Either (Either x y) z -> Either x (Either y z)
30 unassocsum :: Either x (Either y z) -> Either (Either x y) z
```

Fig. 4. Key definitions for `FreerChoiceArrow`. We omit the definitions of `Profunctor`, `Category`, `PreArrow`, and `Arrow` as they are similar to those of `FreerArrow`.

previous form to `Either (_, (_, _)) (_, _)` (line 10). Finally, we apply `left unassoc >>> undistr`, which is the inverse of `distr >>> left assoc`, to the recursive call (line 11). Note that `undistr`, the inverse of `distr`, is defined on line 28.

For the `ChoiceProfunctor` instance, we first apply the `left` method of function arrows to `f`, which gives us a type of form `Either (Either (_, _) _) _`. We then apply the `assocsum` function to convert the result to the form of `Either (_, _) (Either _ _)` (line 15). Finally, we apply `unassocsum`, the inverse of `assocsum` to the recursive call (line 16).

We have even more administrative code to align types in freer choice arrows. In particular, both `distr >>> left assoc` and `left unassoc >>> undistr` in the definition of `first'`, and both `assocsum` and `unassocsum` in the definition of `left'`, pose no computational significance. We talk about another version of freer arrows that avoid these administrative code in the next section.

The `interp` function shows that a freer choice arrow can be interpreted into any choice arrows if provided with an effect handler (lines 18–24). In the case of `Comp`, we apply `first` to `handler x`, just

like in the interprter of `FreerArrows`, but we also apply `left` to the result of `first (handler x)` (line 23). Similar to the interpreter of `FreerArrows`, this is inevitable even for the segments of a freer arrow that we do not use `first` or `left`. We talk about how to remove these unnecessary calls to `first` and `left` in the next section.

Since the freer choice arrow is a sequence of events that might happen depending on the contravariant function in `Comp`, we can only statically over-approximate the number of events that will run in the arrow:

```
overCount :: FreerChoiceArrow e x y -> Int
overCount (Hom _) = 0
overCount (Comp _  _ y) = 1 + overCount y
```

4 BRIDGED FREER ARROWS

There are a few issues with the definitions of freer arrows and freer choice arrows: (1) there are a lot of administrative code for making the types aligned (*e.g.,* in `first'` and `left'`), and (2) the type of the `Comp` constructor forces the interpreter to call `dimap`, `first` (for freer arrows), and `left` (for freer choice arrows) even when it's not necessary.

In this section, we propose bridged freer arrows to resolve these issues with freer arrows and freer choice arrows. The key idea of bridged freer arrows is to *defunctionalize* [Koppel 2019] the function argument in the `Comp` constructor, similar to the approach taken in Chupin and Nilsson [2019]. We call the defunctionalized arguments *bridges*.

4.1 Bridges

We show the definitions of bridges in Fig. 5. A `Bridge` datatype is parameterized by four type parameters (line 2). Intuitively, a bridge describes connections of an effect in a freer arrow with its "previous" and "next" freer arrow component. A bridge of type `Bridge x a b y` connects an input of type `x` to an effect of type `e a b`, and then connects the effect to the "next" freer arrow's input type `y`. Therefore, an `IdBridge` has type `Bridge x x y y`, which means nothing is done before passing `x` to the effect and nothing is done before passing `y` from the effect (line 3). The `FirstBridge` constructor corresponds to the `first'` method of profunctors and the `first` method of arrows: it takes a bridge of type `Bridge x a b y` and turns it into `Bridge (x, c) a b (y, c)` (line 5). `SecondBridge` works similarly (line 6). The `LeftBridge` constructor corresponds to the `left'` method of choice profunctors and the `left` method of arrows: it takes a bridge of type `Bridge x a b y` and turns it into `Bridge (Either x c) a b (Either y c)` (line 8). `RightBridge` works similarly (line 9). Finally, we have `LmapBridge` for functions that do not fall in one of the bridge patterns (line 11).

The function `cmapBridge` is like a smart constructor for `LmapBridge` (lines 13–16). When the bridge is already an `LmapBridge`, it composes the function with the function inside `LmapBridge`. Otherwise, we wrap the original bridge with `LmapBridge`.

Finally, the `bridge` function "interpret"s a `Bridge x a b y` to a function `p a b -> p x y`, where `p` is both a strong profunctor and choice profunctor (lines 18–25).

Bridges are contravariant functors. The Haskell `base` library defines the following typeclass for *contravariant* functors:

```
class Contravariant f where
  contramap :: (a' -> a) -> (f a -> f a')
```

In the bottom part of Fig. 5 (lines 27–32), we show that bridges are in fact a contravariant functor if we switch its arguments. In fact, the definition of its `contramap` method is exactly `cmapBridge`.

```haskell
-- |- Definitions of bridges.
data Bridge x a b y where
  IdBridge :: Bridge x x y y

  FirstBridge :: Bridge x a b y -> Bridge (x, c) a b (y, c)
  SecondBridge :: Bridge x a b y -> Bridge (c, x) a b (c, y)

  LeftBridge  :: Bridge x a b y -> Bridge (Either x c) a b (Either y c)
  RightBridge :: Bridge x a b y -> Bridge (Either c x) a b (Either c y)

  LmapBridge :: (w -> x) -> Bridge x a b y -> Bridge w a b y

-- Bridges are contravariant
cmapBridge :: (w -> x) -> Bridge x a b y -> Bridge w a b y
cmapBridge f (LmapBridge g r) = LmapBridge (g . f) r
cmapBridge f r = LmapBridge f r

-- Bridges transform strong profunctors and choice profunctors
bridge :: (Strong p, Choice p) => Bridge x a b y -> p a b -> p x y
bridge IdBridge = id
bridge (FirstBridge r) = first' . bridge r
bridge (SecondBridge r) = second' . bridge r
bridge (LeftBridge r) = left' . bridge r
bridge (RightBridge r) = right' . bridge r
bridge (LmapBridge f r) = lmap f . bridge r

-- | We don't use this. This is just to show that Router is a contravariant
-- functor.
newtype ContraBridge a b y x = ContraBridge (Bridge x a b y)

instance Contravariant (ContraBridge a b y) where
  contramap f (ContraBridge r) = ContraBridge (cmapBridge f r)
```

Fig. 5. Definitions of bridges.

However, we do not use this typeclass in our development because we like to keep the original ordering of `Bridge`'s type parameters.

Bridges can be made into a profunctor, instead of just a contravariant functor, by adding constructors like `RmapBridge :: Bridge x a b y -> (y -> w) -> Bridge x a b w`. However, we intentionally define bridges in the way they are presented to make them simple. The definition suffices to be used in bridged freer arrows.

4.2 Bridged Freer Arrows

We show the definition of bridged freer arrows in Fig. 6. Comparing with other variants of freer arrows such as `FreerArrow` (Fig. 3) and `FreerChoiceArrow` (Fig. 4), the major difference is that we replace the function argument in the `Comp` constructor with a bridge (line 3).

```
1  data FreerArrow e x y where
2    Hom :: (x -> y) -> FreerArrow e x y
3    Comp :: Bridge x a b z -> e a b ->
4            FreerArrow e z y -> FreerArrow e x y
5
6  embed :: e x y -> FreerArrow e x y
7  embed f = Comp IdBridge f id
8
9  -- Category instance omitted.
10
11 instance Profunctor (FreerArrow e) where
12   dimap f g (Hom h) = Hom $ dimap f g h
13   dimap f g (Comp r x y) = Comp (cmapBridge f r) x (dimap id g y)
14
15   -- lmap can be implemented more efficiently without recursion
16   lmap f (Hom h) = Hom $ lmap f h
17   lmap f (Comp r x y) = Comp (cmapBridge f r) x y
18
19 instance StrongProfunctor (FreerArrow e) where
20   first' (Hom r) = Hom $ first r
21   first' (Comp (LmapBridge f r) a b) =
22     lmap (first f) $ first' (Comp r a b)
23   first' (Comp r a b) =
24     Comp (FirstBridge r) a (first' b)
25
26     -- We also define [second'] separately using [SecondBridge]. Omitted here
27     -- since the definition is similar to [first'].
28
29 instance ChoiceProfunctor (FreerArrow e) where
30    left' (Hom r) = Hom $ left r
31    left' (Comp (LmapBridge f r) a b) =
32      lmap (left f) $ left' (Comp r a b)
33    left' (Comp r a b) =
34      Comp (LeftBridge r) a (left' b)
35
36     -- We also define [right'] separately using [RightBridge]. Omitted here
37     -- since the definition is similar to [left'].
38
39 interp :: (Strong arr, Choice arr, Arrow arr) =>
40   (e :-> arr) -> FreerArrow e x y -> arr x y
41 interp _       (Hom r) = arr r
42 interp handler (Comp f x y) = bridge f (handler x) >>> interp handler y
```

Fig. 6. Definitions of bridged freer arrows.

The `embed` function of bridged freer arrows simply wraps the effect `f` with `IdBridge` inside a `Comp` constructor (lines 6–7).

We omit the definition of bridged freer arrows' `Category` instance as the definition is similar to that of `FreerPreArrows` (Fig. 2). In the `Profunctor` instance, we call the `cmapBridge` function defined in the previous section (Fig. 5) to compose the function with bridges (lines 11–17). In the `StrongProfunctor` instance: if the bridged freer arrow is a `Hom`, we just call the `first` method of function arrows (line 20); if the bridged freer arrow is a `Comp _ a b` with an `LmapBridge f r`, we call the `first` method of function arrows on `f` and make a recursive call to `Comp r a b` (lines 21–22). Otherwise, we simply wrap the original bridge with a `FirstBridge` and makes a recursive call (lines 23–24). In addition, we also define the `second'` method (instead of relying on the default implementation that uses `first'`) so that it uses `SecondBridge`. However, we omit that definition here for conciseness as it is similar to the definition of `first'`. The definition of `ChoiceProfunctor` instance also works similarly (lines 29–37).

Finally, we can define an interpreter for bridged freer arrows using the `bridge` function defined in the previous section (Fig. 5). Comparing with the interpreters of `FreerArrow` and `FreerChoiceArrow`, this interpreter only calls `first'` when there is a `FirstBridge`, only calles `left'` when there is a `LeftBridge`, and only calls `lmap` when there is a `LmapBridge`, *etc.*

Thanks to bridges, the definitions of bridged freer arrows do not contain a heavy amount of administrative code. Additionally, we have slightly more information to use for static analysis, especially when `IdBridge` is used at the top-level in `Comp`.

We can use this extra information to under-approximate the number of events that will run as well as over-approximating them as in freer choice arrows. The implementation of `overCount` remains the same, but we can define the following function for the under-approximation of the count:

```
mightSkip :: Bridge x a b y -> Bool
mightSkip IdBridge = False
mightSkip (FirstBridge b) = mightSkip b
mightSkip (SecondBridge b) = mightSkip b
mightSkip (LeftBridge _) = True
mightSkip (RightBridge _) = True
mightSkip (LmapBridge _ b) = mightSkip b

underCount :: FreerArrow e x y -> Int
underCount (Hom _) = 0
underCount (Comp b _ y) = (if mightSkip b then 0 else 1) + underCount y
```

We use a helper function `mightSkip` to identify bridges which might cause the following event not to run. If the event might not run, we add 0 instead of 1 to the count to get a tight under-approximation.

5 EQUIVALENCE OF FREER ARROWS

Showing that freer arrows can implement methods like `first` is not enough to show that they are *indeed* arrows. We need to show that they satisfy all the arrow laws. For freer pre-arrows, it is straightforward to show that they satisfy all the profunctor laws and pre-arrow laws with respect to Leibniz equality (with the assumption of functional extensionality). However, we cannot do the same for freer arrows or freer choice arrows, because methods like `first` and `left` are too flexible.

In this section, we first discuss why methods like `first` and `left` pose great challenges. We then define new equivalence relations $\approx$ for freer arrows and freer choice arrows that satisfy all the arrow laws and choice arrow laws, respectively. All the definitions, theorems, and proofs described in this section have been formalized in Rocq prover (formerly Coq theorem prover) [Coq development team

2024], but we will use Haskell code throughout this section for consistency. We attach all the Rocq Prover formalizations as supplementary material of this submission and we plan to release them in a publicly available artifact.

5.1 Challenges

To understand challenges imposed by methods like `first` and `left`, let's consider the following law regarding `FreerArrow` and `first`:

```
first f >>> arr fst = arr fst >>> f
```

Here, we use = to represent Leibniz equality.

To prove that freer arrows satisfy this law, we can do an induction on `f`:

In the base case, we would like to show that `first (Hom y) >>> arr fst = arr fst >>> Hom y` for some function `y`. The expression on both sides evaluate to a `Hom`, so it suffices to show that the functions inside `Hom` are equivalent.

In the inductive case, we assume `first f >>> arr fst = arr fst >>> f`. We would like to show that `first (Comp p e a) >>> arr fst = arr fst >>> Comp p e a` under the following typing context:

```
p :: x -> (a, c)
e :: e a b
a :: FreerArrow e (b, c) y
```

The expression on the left hand side of the equality evaluates to:

```
Comp (first p >>> assoc)
     e
     (lmap unassoc (first' a) >>> arr fst)
```

The expression on the right hand side of the equality evaluates to:

```
Comp (fst >>> p) e a
```

Normally, to show equality at this step, we want to establish that all the arguments on both sides' `Comp` are equal. However, that is not possible here because the types of these arguments on both sides do not even match! The function argument on the left hand side of the equality has type `(x, w) -> (a, (c, w))` for some existential type `w`. The function argument on the right hand side has type `(x, w) -> (a, c)`. Correspondingly, the inner freer arrow on the left hand side has type `FreerArrow e (b, (c, w)) y`. The inner freer arrow on the right hand side has type `FreerArrow e (b, c) y`. At this point, we have to give up the proof.

Similarly, it is also impossible to prove the following law regarding `FreerChoiceArrow` and the `left` method:

```
f >>> arr Left = arr Left >>> left f
```

Again, we try to prove this by induction on `f`:

In the base case, we need to prove that `Hom y >>> arr Left = arr Left >>> left (Hom y)`. The expression on both sides evaluate to a `Hom`, so it suffices to show that the functions inside `Hom` are equivalent.

In the inductive case, we assume that `f >>> arr Left = arr Left >>> left f`. We want to show that `Comp p e x >>> arr Left = arr Left >>> left (Comp p e x)` under the following typing context:

```
p :: x -> Either (a, c) w
e :: e a b
```

```
a :: FreerChoiceArrow e (Either (b, c) w) y
```

The expression on the left hand side of the equality evaluates to:

```
Comp p e (f >>> arr Left)
```

The expression on the right hand side of the equality evaluates to:

```
Comp (Left >>> left p >>> assocsum)
     a
     (lmap unassocusum (left' a))
```

Unfortunately, we have again run into a case of type mismatch. The function argument on the left hand side has type `x -> Either (a, c) w`. The function argument on the right hand side has type `x -> Either (a, c) (Either w w')` for some existential type `w'`. Correspondingly, the inner freer arrow on the left hand side has type `FreerChoiceArrow e (Either (b, c) w) y`. The inner freer arrow on the right hand side has type `FreerChoiceArrow e (Either (b, c) (Either w w')) y`. Again, we are at an impasse.

These failed proofs show that we need an equivalence relation that can deal with the discrepancy of existential variables inside `Comp`s. Our key observation, however, is that *some of these existential variables do not matter in the end.* We discuss a novel equivalence relation for freer arrows and freer choice arrows in the next two sections.

5.2 Equivalence for Freer Arrows

The equivalence relation between freer arrows should simultaneously capture two things: (1) the equivalence of all the function arguments at every level "joined" together (the "semantic" part), and (2) the similarity between freer arrow structures (the "syntactic" part). For (1), we propose a dependently-typed characteristic function $C(\cdot)$ that represents recursively "joining" all the function arguments in a freer arrow. For (2), we propose a similarity relation $\sim$ that only capture structural similarity between freer arrows.

Characteristic functions. We show the typing rules and definitions of characteristic functions in the top part of Fig. 7. A characteristic function is dependently typed, as its return type depends on the value of its first argument, *i.e.,* the freer arrow.

The typing rule HomType states that the characteristic type of `Hom` f is the return type of f. The CompType rule is defined inductively: In the case of `Comp` f e y, the characteristic function "collects" A, the input type of e, and constructs a function that takes B, the output type of e, as input and returns Z, the characteristic type of y. Note that the `Comp` constructor contains three existential types represented by A, B, and C in the typing rules, but only A and B appear in the characteristic function type. This is because C does not matter *in the end*: only A and B are directly related to the effect e and C does not appear anywhere else. Ignoring C is crucial for characteristic functions, as this allows us to equate two freer arrows with existential types differ in C.

Characteristic functions are defined recursively. In the base case `Hom` f, the characteristic function is just f. In the case of `Comp` f e y, the characteristic function is the "join" of f and $C(y)$, where the "join" operation is formally defined by the $\bowtie$ operator. The $\bowtie$ operator takes two functions f and g of types $(X \to A \times C)$ and $(B \times C \to Z)$, respectively. It constructs a new function of type $X \to (A \times (B \to Z))$ that, given an input x of type X, it collects the value of type A produced by applying f to x and a function of type $B \to Z$ using g and a value c of type C produced by applying f to x. After using c in the function application of g, it is never needed again, so its type disappears from the return value of the characteristic function.

Characteristic Function Type $\mathcal{T}(\cdot)$:

$$\frac{\Gamma \vdash \mathtt{f} : X \to Y}{\Gamma \vdash \mathcal{T}(\mathtt{Hom}\ f) = Y}\textsc{HomType}$$

$$\frac{\Gamma \vdash \mathtt{f} : X \to A \times C \qquad \Gamma \vdash \mathtt{e} : E\ A\ B \qquad \Gamma \vdash \mathtt{y} : \mathtt{FreerArrow}\ E\ (B \times C)\ Y \qquad \Gamma \vdash \mathcal{T}(y) = Z}{\Gamma \vdash \mathcal{T}(\mathtt{Comp}\ f\ e\ y) = A \times (B \to Z)}\textsc{CompType}$$

Characteristic Function $C(\cdot)$:

$$C :: (f :: \mathtt{FreerArrow}\ E\ X\ Y) \to X \to \mathcal{T}(f)$$
$$C(\mathtt{Hom}\ f) = f$$
$$C(\mathtt{Comp}\ f\ e\ y) = f \bowtie C(y)$$
$$\text{where } \bowtie :: (X \to A \times C) \to (B \times C \to Z) \to X \to (A \times (B \to Z))$$
$$f \bowtie g = \lambda\, x.\mathrm{let}\ (a, c) = f\ x\ \mathrm{in}\ (a, \lambda\, b.g\ (b, c))$$

Similarity $\sim$ between `FreerArrows`:

$$\frac{}{\mathtt{Hom}\ f \sim \mathtt{Hom}\ g}\textsc{HomSimilar} \qquad \frac{x \sim y}{\mathtt{Comp}\ f\ e\ x \sim \mathtt{Comp}\ g\ e\ y}\textsc{CompSimilar}$$

Equivalence $\approx$ between `FreerArrows`:

$$\frac{x \sim y \qquad C(x) = C(y)}{x \approx y}\textsc{ArrowEq}$$

Fig. 7. Characteristic functions, similarity $\sim$, and equivalence $\approx$ of `FreerArrows`.

Similarity. We show the similarity relation $\sim$ between freer arrows in Fig. 7. The HomSimilar rule states that `Hom` f and `Hom` g are always similar regardless of the relation between f and g. The CompSimilar rule states that, if x and y are similar, `Comp` $f\ e\ x$ and `Comp` $g\ e\ y$ are similar. The rule does not require any relations between f and g, but it does require both `Comps` to have the same e. Intuitively, two freer arrows are similar as long as they share the same structure, *i.e.,* same number of `Comps`, and the same effects.

The similarity relation is heterogeneous, as it does not need both operands to share the same type. However, we can prove the following lemma if both operands share the same type:

Lemma 5.1. *If both operands share the same type, $\sim$ is an equivalence relation, i.e., it is reflexive, symmetric, and transitive.*

Proof. The proofs of reflexivity and symmetry are straightforward by induction. However, for transitivity, we need to prove a more general theorem on operands that don't share the same type:

$$\forall(a :: \mathtt{FreerArrow}\ e\ x\ r)(b :: \mathtt{FreerArrow}\ e\ y\ r)(c :: \mathtt{FreerArrow}\ e\ z\ r), a \sim b \implies b \sim c \implies a \sim c.$$

This more general theorem can be proven by induction over $x \sim y$. It then implies the more restrictive transitivity theorem we want. □

There is also an important relation between characteristic function types and similarity:

Theorem 5.2. *For all freer arrows x and y, if $x \sim y$, then $\mathcal{T}(x) = \mathcal{T}(y)$.*

Proof. By induction over $x \sim y$. □

Equivalence. Finally, equivalence $\approx$ is defined in terms of characteristic functions and $\sim$, shown in the bottom of Fig. 7. Note that we can directly compare $C(x)$ and $C(y)$ because $x \sim y$ implies $\mathcal{T}(x) = \mathcal{T}(y)$ by Theorem 5.2.

Lemma 5.3. *$\approx$ is an equivalence relation, i.e., it is reflexive, symmetric, and transitive.*

Proof. We can prove this by induction with the help of Lemma 5.1 and Theorem 5.2. □

Theorem 5.4 (Freer arrows are lawful arrows). *The `FreerArrow` datatype satisfies all the profunctor laws, pre-arrow laws, and arrow laws (Appendix A) with respect to $\approx$.*

Proof. We can prove that all the profunctor laws and arrow laws except for three laws hold for the stronger Leibniz equality. Therefore, we prove these laws using the Leibniz equality, which implies $\approx$. All of these laws can be proven either directly by definition or by induction over a freer arrow.

For the following three laws, we directly prove them under $\approx$:

```
first f >>> arr fst ≈ arr fst >>> f
first f >>> arr (id *** g) ≈ arr (id *** g) >>> first f
first (first f) >>> arr assoc ≈ arr assoc >>> first f
```

All these three laws can be proven by an induction over the freer arrow `f`. □

We formalize all these definitions, theorems, and proofs in Rocq Prover. Because the characteristic function type $\mathcal{T}(f)$ of a freer arrow f depend on the structure of f, we use dependent types in formalizing characteristic functions. This makes it difficult to state the definition ArrowEq, because we need to prove that $C(x)$ and $C(y)$ share the same type (*i.e.,* $\mathcal{T}(x) = \mathcal{T}(y)$) before stating that $C(x) = C(y)$. The type equality can be proven by applying Theorem 5.2, but working with this definition with type casting still requires significant proof engineering. Our Rocq proof assumes the axiom of functional extensionality and the axiom of unicity of identity proofs. We also rely on techniques about reasoning about equality proofs presented in Chlipala [2013]. More detail about our Rocq formalization can be found in Appendix B and in our Rocq development.

5.3 Freer Choice Arrows

The idea of characteristic functions, $\sim$, and $\approx$ applies to freer choice arrows as well. However, we need to overload the definition of characteristic functions and their types to accomondate the extra capability of freer choice arrows. We show the overloaded definitions of $\mathcal{T}(\cdot)$ and $C(\cdot)$ in Fig. 8. We also need to overload $\sim$ and $\approx$ for freer choice arrows and to use the new characteristic functions, but we omit these definitions here as they are almost the same as those of freer arrows.

We can prove lemmas similar to Lemma 5.1–5.3 for freer choice arrows as well, the proofs are similar so we omit them here. These lemmas lead us to the following theorem:

Theorem 5.5 (Freer choice arrows are lawful choice arrows). *The `FreerChoiceArrow` datatype satisfies all the profunctor laws, pre-arrow laws, arrow laws, and choice arrow laws (Appendix A) with respect to $\approx$.*

Proof. Similar to the proof of Theorem 5.4, we can prove most laws using Leibniz equality. We need $\approx$ for the following laws:

Characteristic Function Types $\mathcal{T}(\cdot)$:

$$\frac{\Gamma \vdash \mathtt{f} : X \to Y}{\Gamma \vdash \mathcal{T}(\mathtt{Hom}\ f) = Y}\textsc{HomType}$$

$$\frac{\Gamma \vdash \mathtt{f} : X \to A \times C + W \qquad \Gamma \vdash \mathtt{e} : E\ A\ B \qquad \Gamma \vdash \mathtt{y} : \mathtt{FreerChoiceArrow}\ E\ (B \times C + W)\ Y \qquad \Gamma \vdash \mathcal{T}(y) = Z}{\Gamma \vdash \mathcal{T}(\mathtt{Comp}\ f\ e\ y) = A \times (B \to Z) + Z}\textsc{CompType}$$

Characteristic Functions $C(\cdot)$:

$$C :: (f :: \mathtt{FreerChoiceArrow}\ E\ X\ Y) \to X \to \mathcal{T}(f)$$
$$C(\mathtt{Hom}\ f) = f$$
$$C(\mathtt{Comp}\ f\ e\ y) = f \bowtie C(y)$$
$$\text{where } \bowtie :: (X \to A \times C + W) \to (B \times C + W \to Z) \to X \to (A \times (B \to Z) + Z)$$
$$f \bowtie g = \lambda\, x.\mathrm{case}\ (f\ x)\ \mathrm{of}$$
$$\mathrm{Left}\ (a, c) \Rightarrow \mathrm{Left}\ (a, \lambda\, b.g\ (\mathrm{Left}\ (b, c)))$$
$$\mathrm{Right}\ w \Rightarrow \mathrm{Right}\ (g\ (\mathrm{Right}\ w))$$

Fig. 8. Characterstic functions of `FreerChoiceArrows`.

```
first f >>> arr fst ≈ arr fst >>> f
first f >>> arr (id *** g) ≈ arr (id *** g) >>> first f
first (first f) >>> arr assoc ≈ arr assoc >>> first f
f >>> arr Left ≈ arr Left >>> left f
left f >>> arr (id +++ g) ≈ arr (id +++ g) >>> left f
left (left f) >>> arr assocsum ≈ arr assocsum >>> left f
```

The first three laws are the same laws that we prove for `FreerArrows` using ≈. The last three laws are unique to choice arrows. All of these six laws can be proven by induction over the freer choice arrow `f`. □

5.4 Bridged Freer Arrows

We do not define an equivalence relation for bridged freer arrows. Instead, we define the following "translation" from bridged freer arrows to freer choice arrows:

```
translate :: BridgedFreerArrow e x y -> FreerChoiceArrow e x y
translate = B.interp F.embed
```

The `B.interp` function stands for the `interp` function in the module of briged freer arrows. The `F.embed` function stands for the `embed` function in the module of freer choice arrows.

We can then use the equivalence relation of freer choice arrows to equate bridged freer arrows.

6 CASE STUDY: THE STATE ARROW

In this section, we use a simple state arrow to demonstrate the usefulness of freer arrows. We will describe a larger case study in the next section.

We define a typeclass `ArrowState` that describes arrows that include a state effect (similar to the `MonadState` in Haskell's `mtl` library [Jones 1995]):

```
class Arrow a => ArrowState s a where
  get :: a b s
  put :: a s s
```

For convenience, the `get` operation has an input type `b` which allows it to consume *any* input, and `put` outputs the value written so that it can be further used.

We can define a type of effects `StateEff` which describes the same operations:

```
-- |- An ADT for stateful effect.
data StateEff :: Type -> Type -> Type -> Type where
  Get :: StateEff s a s
  Put :: StateEff s s s
```

We can then declare a freer arrow as an `ArrowState` as follows:

```
-- |- A freer arrow is an ArrowState.
instance ArrowState s (FreerArrow (StateEff s)) where
  get = embed Get
  put = embed Put

handleState :: ArrowState s a => StateEff s x y -> a x y
handleState Get = get
handleState Put = put
```

Similarly, we can show that freer choice arrows are an instance of `ArrowState` as well.

Effects are composable. Here, we only show freer arrows with state effects for simplicity, but it is possible to define methods for freer arrows with any effects as long as the effects *include* state effects. The method for defining composable effects is well-studied in existing literatures on algebraic effects and effect handlers, *etc.*, so we defer the detailed implementation based on these literatures to Appendix C.

Examples. We can now use the `ArrowState` interface to define a freer arrow. For example, the following function `inc n` starts from an initial number and keeps increasing the state for `n` times:

```
inc :: Int -> FreerArrow (StateEff Int) Int Int
inc n | n > 0     = get >>> lmap (+1) put >>> inc (n - 1)
      | otherwise = get
```

We can then either interpret the freer arrow to a state arrow, or an **`IO`** arrow, or we can do some static analysis on the freer arrow, *e.g.,* count the number of `get`s and `put`s.

The `inc` example shows that we can write programs using the `ArrowState` interface and freer arrows to define recursions where the depth of recursion is statically known. What if we want to write programs with loops where the number of iterations is not statically known?

While freer arrows are not typically for this type of tasks, we can combine freer arrows with some iteration combinators to implement this. For example, we show an `Elgot` datatype that represents the Elgot iteration [Adámek et al. 2011; Goncharov 2022] in Fig. 9. `Elgot` takes three parameters `e :: Type -> Type -> Type`, `x :: Type`, and `y :: Type`. It has only one constructor `Elgot` that takes a "loop body" of type `f e x (Either z x)` and a continuation of type `f e z y`. The intuition is that if the loop body returns a value of type `x`, the loop body will be executed again. Otherwise, we end the loop and run the continuation instead. The logic is also reflected by the `interp` method of `Elgot`.

With the `Elgot` datatype, we can define a countdown function using freer choice arrows:

```
countA :: Elgot FreerChoiceArrow (StateEff Int) Int Int
countA =
```

```
data Elgot f (e :: Type -> Type -> Type) x y where
  Elgot :: f e x (Either z x) -> f e z y -> Elgot f e x y

interp :: ArrowChoice arr =>
  (f e :-> arr) -> Elgot f e x y -> arr x y
interp h (Elgot l k) =
  let go = h l >>> h k ||| go in go
```

Fig. 9. A datatype representing the Elgot iteration [Adámek et al. 2011; Goncharov 2022].

```
class Arrow a => ArrowMapState k v a where
  getM :: a k v
  putM :: a (k, v) v

class Arrow a => ArrowIndexedMapState k v a where
  getIM :: k -> a b v
  putIM :: k -> a v v

-- |- An ADT for stateful effect.
data MapStateEff :: Type -> Type -> Type -> Type -> Type where
  GetM :: MapStateEff k v k v
  PutM :: MapStateEff k v (k, v) v

data IndexedMapStateEff :: Type -> Type -> Type -> Type -> Type where
  GetIM :: k -> IndexedMapStateEff k v a v
  PutIM :: k -> IndexedMapStateEff k v v v

instance ArrowMapState k v (FreerArrow (MapStateEff k v)) where
  getM = embed GetM
  putM = embed PutM

instance ArrowIndexedMapState k v (FreerArrow (IndexedMapStateEff k v)) where
  getIM k = embed $ GetIM k
  putIM k = embed $ PutIM k
```

Fig. 10. Two arrow interfaces for a simple key-value store.

```
  let go = get >>> arr (\n -> if n == 0 then Left n else Right n) >>>
           right (lmap (\x -> x - 1) put) in
    Elgot go id
```

A simple key-value store. We can extend the state arrow to a simple key-value store. There are two ways we can do this with arrows: keys can be passed as an input to the event in the arrow or as a functional parameter to an event. Therefore, we get two different key-value store interfaces, shown as `ArrowMapState` and `ArrowIndexedMapState` in Fig. 10.

```
putput :: Eq k => FreerArrow (IndexedMapStateEff k v) a b ->
          FreerArrow (IndexedMapStateEff k v) a b
putput (Comp f (PutIM k1) (Comp IdBridge (PutIM k2) c))
  | k1 == k2  = Comp f (PutIM k1) c
  -- otherwise, fall through; writes to different keys still need to happen
putput (Comp f e c) = Comp f e $ putput c
putput x = x
```

Fig. 11. A simple optimization that can be performed the bridged freer arrow version of the key-value store. More optimizations can be found in Appendix D.

A bridged freer arrow with `IndexedMapStateEff` can be used to implement some optimizations following algebraic properties of a key-value map. We show one in Fig. 11 and defer the rest of them to Appendix D. These optimizations are given as an example and are very narrow. For example they don't traverse through `FirstBridge` or `LeftBridge` to find places to optimize.

Unfortunately, it is less obvious how to define these for `MapStateEff` where the key is data in the arrow. The bridges we have are not sufficient for detecting when the same key is used twice.

7 CASE STUDY: CHOREOGRAPHIC PROGRAMMING

HasChor [Shen et al. 2023] is a library for choreographic programming which uses freer monads to encode programs that run choreographically on multiple physical or logical machines, or *endpoints*. A key feature of HasChor is that you only need write the program *once* and have the code distributed to all the endpoints. Thanks to freer monads, a user can re-use all existing advanced features from the host language, Haskell, including higher-order functions, type systems, *etc.* In this section, we present HasChorA, a larger case study that adapts HasChor to use freer arrows instead. We also work out an example of the Diffie-Hellman key exchange protocol included with the HasChor library.

When using HasChorA, a user first writes the single choreography (*i.e.,* the program to be distributed among multiple endpoints) using freer choice arrows and the following effect datatype:

```
data ChoreoSig ar b a where
  Local :: KnownSymbol l => Proxy l -> ar b a ->
           ChoreoSig ar (b @ l) (a @ l)

  Comm :: (Show a, Read a, KnownSymbol l, KnownSymbol l') =>
          Proxy l -> Proxy l' ->
          ChoreoSig ar (a @ l) (a @ l')

  Cond :: (Show a, Read a, KnownSymbol l) =>
          Proxy l -> Choreo ar a (Either b c) ->
          ChoreoSig ar (a @ l) (Either b c)

type Choreo ar = FreerChoiceArrow (ChoreoSig ar)
```

These definitions use a special datatype called locations that indicate at which endpoint the data is. Events in the `Choreo` arrows are `Local`, for computations that run locally at one location (the locations between the input and output are the same); `Comm`, for sending a value located at one

location to another (the locations between the input and output differ); and `Cond`, for conditionally choosing what to do next based on a located value.

The choregrahphy can then be projected into different endpoints. At each endpoint, a set of network effects that only specifies what this particular endpoint does is used. For example, a `Comm` effect may become a "send" if the endpoint's location matches the input's location, or a "receive", if it matches the output's location, or simply discarded, if it matches no location. These network effects in HasChorA are defined as follows:

```
data NetworkSig ar b a where
  Run :: ar b a -> NetworkSig ar b a             -- Local computation.
  Send :: Show a => LocTm -> NetworkSig ar a ()  -- Sending.
  Recv :: Read a => LocTm -> NetworkSig ar () a  -- Receiving.
  BCast :: Show a => NetworkSig ar a ()          -- Broadcasting.

type Network ar = FreerChoiceArrow (NetworkSig ar)
```

Static analysis (*e.g.*, counting the number of `Send` and `Recv` operations) and optimizations (*e.g.*, combining sequential `Comm` or `Send`/`Recv` events) may be performed on the program. It would be difficult or infeasible to do with the original freer monad version without metaprogramming. For example, we can count the number of each kind of `Send` and `Recv` events in the `Network` arrow using the following function:

```
count_send_recv :: Network ar a b -> (Integer, Integer)
count_send_recv (Comp _ e k) =
  let (s, r) = count_send_recv k in
    case e of
      Send _ -> (s + 1, r)
      Recv _ -> (s, r + 1)
      _ -> (s, r)
count_send_recv (Hom _) = (0, 0)
```

8 RELATED WORK

Arrows. Hughes [2000] initially proposes arrows as a generalization of monads. However, the relation among monads, applicative functors (also known as idioms at that time) [McBride and Paterson 2008], and arrows were unknown at that time. It was only later shown by Lindley et al. [2008] that arrows are between applicative functors and monads in terms of the expressiveness. Similar to monads, a **do**-notation for arrows can also be used in writing arrows, instead of using primitive methods like `first` and `left` [Paterson 2001, 2003].

Arrows have been commonly used in domains like functional reactive programming [Chupin and Nilsson 2019; Hudak et al. 2003; Keating and Gale 2024; Perez et al. 2016]. For example, Hudak et al. [2003] show that arrows are useful for encoding behaviors in robots that combine continuous and discrete parts, such as integration or derivation of sensor signals and changing between finite program states. Chupin and Nilsson [2019] propose the concept of routers that defunctionalizes the connections between arrows, which are direct inspirations of our bridged version of freer arrows. However, the use of our bridges are narrower than their routers because we would like to keep bridges as simple as possible. In the future, we would like to explore with routers more extensively to study optimizing freer arrows.

In other domains, Carette et al. [2024] use arrows to represent quantum computations as classical computations. Notably, in this computation model, the underlying classical language is restricted to

reversible functions, unlike the unrestricted pure functions in the Haskell version. This brings up an interesting question about one of the defining arrow operators `arr`, which normally lifts a "pure" function into the arrow. We leave it to future work to consider what notions of pure functions can be lifted into an arrow, and whether it generalizes to, for example, any underlying category or profunctor.

Free structures. We discussed freer monads and their usefulness in Section 1. Other freer structures have also been studied. For example, Capriotti and Kaposi [2014] propose a version of freer applicative functors and show that they can be statically analyzed. Mokhov et al. [2019] propose freer selective applicative functors, which have been used by Willis et al. [2020] to implement efficient staged parser combinators. Rivas and Jaskelioff [2017] study various free strutures, including free monads, free applicative functors, and free arrows. By applying the techniques used by Kiselyov and Ishii [2015] to derive freer monads, we can derive freer versions of these structures—in fact, we initially derive our versions of freer arrows in this way.

Free structures enable a mixed embedding that has both semantics parts and syntactic parts in the same data structure. This use case of free structures has been studied by various works [Chlipala 2021; Gibbons and Wu 2014; Korkut et al. 2025; Li and Weirich 2022].

Algebraic effects with arrows. More recently, Sanada [2024] describes a semantics for writing and interpreting effect handlers using arrows. This is parallel to our work, because it involves defining a language with a structure for writing user-defined effect handlers in that language. Our work does not involve a bespoke language with effects, but rather describes a definition within an existing language (Haskell) for freer arrows, along with tooling to use it for extensible effects.

9 CONCLUSION AND FUTURE WORK

In this paper, we define and study the four variants of freer arrows: freer pre-arrows, freer arrows, freer choice arrows, and bridged freer arrows. We develop a novel equivalence relation to show that these freer arrows are lawful. The equivalence relation is based on a characteristic function that capture the semantic parts of freer arrows, and a similarity relation that capture the syntactic parts of freer arrows. These freer arrows are amenable to static analysis. In particular, we can count the exact number of effect occurances in freer pre-arrows and freer arrows; we can get an over-approximation in freer choice arrows; we can get both an under-approximation and an over-approximation in bridged freer arrows. We conducted case studies of freer arrows based on a simple key/value store and a choreographic programming library HasChor. These case studies show that freer arrows are expressive to be used in practice.

In the future, we would like to look into more complex optimizations based on freer arrows. One challenge with such optimizations is that the function components in freer arrows are hard to analyze. One possibility is defunctionalize commonly used "routing" functions, similar to the approach of Chupin and Nilsson [2019]. We would also like to apply arrows to other libraries that make use of static structures, such as Haxl [Marlow et al. 2014] and build systems [Mokhov et al. 2020].

DATA-AVAILABILITY STATEMENT

We include all the Haskell and Rocq Prover development for this paper, including the definitions of variants of freer arrows, case studies, and formal proofs, in the supplementary materials of this submission. We intend to submit all the code (with more comprehensive and clear comments and instructions) as a publicly-available artifact during artifact evaluation.

REFERENCES

Jirí Adámek, Stefan Milius, and Jirí Velebil. 2011. Elgot theories: a new perspective on the equational properties of iteration. *Math. Struct. Comput. Sci.* 21, 2 (2011), 417–480. https://doi.org/10.1017/S0960129510000496

Kazuyuki Asada. 2010. Arrows Are Strong Monads. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010*, Venanzio Capretta and James Chapman (Eds.). ACM, 33–42. https://doi.org/10.1145/1863597.1863607

Kazuyuki Asada and Ichiro Hasuo. 2010. Categorifying Computations into Components via Arrows as Profunctors. In *Proceedings of the Tenth Workshop on Coalgebraic Methods in Computer Science, CMCS@ETAPS 2010, Paphos, Cyprus, March 26-28, 2010 (Electronic Notes in Theoretical Computer Science, Vol. 264)*, Bart Jacobs, Milad Niqui, Jan J. M. M. Rutten, and Alexandra Silva (Eds.). Elsevier, 25–45. https://doi.org/10.1016/J.ENTCS.2010.07.012

Venanzio Capretta. 2005. General recursion via coinductive types. *Log. Methods Comput. Sci.* 1, 2 (2005). https://doi.org/10.2168/LMCS-1(2:1)2005

Paolo Capriotti and Ambrus Kaposi. 2014. Free Applicative Functors. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Levy and Neel Krishnaswami (Eds.). 2–30. https://doi.org/10.4204/EPTCS.153.2

Jacques Carette, Chris Heunen, Robin Kaarsgaard, and Amr Sabry. 2024. How to Bake a Quantum Π. *Proc. ACM Program. Lang.* 8, ICFP (Aug. 2024), 236:1–236:29. https://doi.org/10.1145/3674625

Adam Chlipala. 2013. Reasoning About Equality Proofs. In *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press. http://mitpress.mit.edu/books/certified-programming-dependent-types

Adam Chlipala. 2021. Skipping the binder bureaucracy with mixed embeddings in a semantics course (functional pearl). *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–28. https://doi.org/10.1145/3473599

Jan Christiansen, Sandra Dylus, and Niels Bunkenburg. 2019. Verifying effectful Haskell programs in Coq. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, Richard A. Eisenberg (Ed.). ACM, 125–138. https://doi.org/10.1145/3331545.3342592

Guerric Chupin and Henrik Nilsson. 2019. Functional Reactive Programming, restated. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 7:1–7:14. https://doi.org/10.1145/3354166.3354172

Allele Dev, Ixcom Core Team, Alexis King, and other contributors. 2024. freer-simple: A friendly effect system for Haskell. https://hackage.haskell.org/package/freer-simple-1.2.1.2

Sandra Dylus, Jan Christiansen, and Finn Teegen. 2019. One Monad to Prove Them All. *Art Sci. Eng. Program.* 3, 3 (2019), 8. https://doi.org/10.22152/programming-journal.org/2019/3/8

Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: deep and shallow embeddings (functional Pearl). In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 339–347. https://doi.org/10.1145/2628136.2628138

Sergey Goncharov. 2022. Shades of Iteration: From Elgot to Kleene. In *Recent Trends in Algebraic Development Techniques - 26th IFIP WG 1.3 International Workshop, WADT 2022, Aveiro, Portugal, June 28-30, 2022, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13710)*, Alexandre Madeira and Manuel A. Martins (Eds.). Springer, 100–120. https://doi.org/10.1007/978-3-031-43345-0_5

Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming, 4th International School (Lecture Notes in Computer Science, Vol. 2638)*, Johan Jeuring and Simon Peyton Jones (Eds.). Springer-Verlag. http://www.haskell.org/yale/papers/oxford02/

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

Bart Jacobs, Chris Heunen, and Ichiro Hasuo. 2009. Categorical semantics for arrows. *J. Funct. Program.* 19, 3-4 (2009), 403–438. https://doi.org/10.1017/S0956796809007308

Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text (Lecture Notes in Computer Science, Vol. 925)*, Johan Jeuring and Erik Meijer (Eds.). Springer, 97–136. https://doi.org/10.1007/3-540-59451-5_4

Finnbar Keating and Michael B. Gale. 2024. Functional Reactive Programming, Rearranged. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium, Haskell 2024, Milan, Italy, September 6-7, 2024*, Niki Vazou and J. Garrett Morris (Eds.). ACM, 55–67. https://doi.org/10.1145/3677999.3678278

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*. 94–105. https://doi.org/10.1145/2804302.2804319

James Koppel. 2019. Defunctionalization: Everybody Does It, Nobody Talks About It. *ACM SIGPLAN PL Perspectives* (2019). https://blog.sigplan.org/2019/12/30/defunctionalization-everybody-does-it-nobody-talks-about-it/

Joomy Korkut, Kathrin Stark, and Andrew W. Appel. 2025. A Verified Foreign Function Interface between Coq and C. *Proc. ACM Program. Lang.* 9, POPL (2025), 687–717. https://doi.org/10.1145/3704860

Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. 2018. Modular Verification of Programs with Effects and Effect Handlers in Coq. In *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 15-17, 2018, Proceedings*. 338–354. https://doi.org/10.1007/978-3-319-95582-7_20

Yishuai Li, Benjamin C. Pierce, and Steve Zdancewic. 2021. Model-based testing of networked applications. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 529–539. https://doi.org/10.1145/3460319.3464798

Yao Li and Stephanie Weirich. 2022. Program adverbs and Tlön embeddings. *Proc. ACM Program. Lang.* 6, ICFP (2022), 312–342. https://doi.org/10.1145/3547632

Sam Lindley, Philip Wadler, and Jeremy Yallop. 2008. Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous. In *Proceedings of the Second Workshop on Mathematically Structured Functional Programming, MSFP@ICALP 2008, Reykjavik, Iceland, July 6, 2008 (Electronic Notes in Theoretical Computer Science, Vol. 229)*, Venanzio Capretta and Conor McBride (Eds.). Elsevier, 97–117. https://doi.org/10.1016/J.ENTCS.2011.02.018

Sandy Maguire. 2025. polysemy: Higher-order, low-boilerplate free monads. https://hackage.haskell.org/package/polysemy-1.9.2.0

Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: an abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 325–337. https://doi.org/10.1145/2628136.2628144

Coq development team. 2024. *The Coq proof assistant*. https://doi.org/10.5281/zenodo.14542673

Conor McBride. 2015. Turing-Completeness Totally Free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*. 257–275. https://doi.org/10.1007/978-3-319-19797-5_13

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *J. Funct. Program.* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jerémie Dimino. 2019. Selective applicative functors. *Proc. ACM Program. Lang.* 3, ICFP (2019), 90:1–90:29. https://doi.org/10.1145/3341694

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2020. Build systems à la carte: Theory and practice. *J. Funct. Program.* 30 (2020), e11. https://doi.org/10.1017/S0956796820000088

Ross Paterson. 2001. A New Notation for Arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 229–240. https://doi.org/10.1145/507635.507664

Ross Paterson. 2003. Arrows and Computation. In *The Fun of Programming*, Jeremy Gibbons and Oege de Moor (Eds.). Palgrave, 201–222. http://www.soi.city.ac.uk/~ross/papers/fop.html

Ivan Perez, Manuel Bärenz, and Henrik Nilsson. 2016. Functional reactive programming, refactored. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, Geoffrey Mainland (Ed.). ACM, 33–44. https://doi.org/10.1145/2976002.2976010

Maciej Piróg and Jeremy Gibbons. 2014. The Coinductive Resumption Monad. In *Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2014, Ithaca, NY, USA, June 12-15, 2014 (Electronic Notes in Theoretical Computer Science, Vol. 308)*, Bart Jacobs, Alexandra Silva, and Sam Staton (Eds.). Elsevier, 273–288. https://doi.org/10.1016/j.entcs.2014.10.015

Exequiel Rivas and Mauro Jaskelioff. 2017. Notions of computation as monoids. *J. Funct. Program.* 27 (2017), e21. https://doi.org/10.1017/S0956796817000132

Takahiro Sanada. 2024. Algebraic effects and handlers for arrows. *J. Funct. Prog.* 34 (2024), e8. https://doi.org/10.1017/S0956796824000066

Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP (2023), 541–565. https://doi.org/10.1145/3607849

Lucas Silver, Paul He, Ethan Cecchetti, Andrew K. Hirsch, and Steve Zdancewic. 2023. Semantics for Noninterference with Interaction Trees. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:29. https://doi.org/10.4230/LIPICS.ECOOP.2023.29

Lucas Silver and Steve Zdancewic. 2021. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28. https://doi.org/10.1145/3434307

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.* 4, ICFP (2020), 121:1–121:30. https://doi.org/10.1145/3409003

Philip Wadler. 1992. Comprehending Monads. *Math. Struct. Comput. Sci.* 2, 4 (1992), 461–493. https://doi.org/10.1017/S0960129500001560

Jamie Willis, Nicolas Wu, and Matthew Pickering. 2020. Staged selective parser combinators. *Proc. ACM Program. Lang.* 4, ICFP (2020), 120:1–120:30. https://doi.org/10.1145/3409002

Nicolas Wu, Tom Schrijvers, Rob Rix, and Patrick Thomson. 2025. fused-effects: A fast, flexible, fused effect system. https://hackage.haskell.org/package/fused-effects-1.1.2.4

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

Kangfeng Ye, Simon Foster, and Jim Woodcock. 2022. Formally Verified Animation for RoboChart using Interaction Trees. In *The 23rd International Conference on Formal Engineering Methods*. Springer Science and Business Media Deutschland GmbH. https://eprints.whiterose.ac.uk/188566/

Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 2021. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference) (LIPIcs, Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:19. https://doi.org/10.4230/LIPIcs.ITP.2021.32

A APPENDIX: ARROW LAWS

```
-- Profunctor laws
dimap (f . g) (h . i) ≈ dimap g h . dimap f i
lmap (f . g) ≈ lmap g . lmap f
rmap (f . g) ≈ rmap f . rmap g
-- Category laws
id >>> f ≈ f
f >>> id ≈ f
(f >>> g) >>> h ≈ f >>> (g >>> h)
-- Pre-Arrow laws
arr id ≈ id
arr (f >>> g) ≈ arr f >>> arr g
-- Arrow laws
first (arr f) ≈ arr (first f)
first (f >>> g) ≈ first f >>> first g
first f >>> arr fst ≈ arr fst >>> f
first f >>> arr (id *** g) ≈ arr (id *** g) >>> first f
first (first f) >>> arr assoc ≈ arr assoc >>> first f
-- Choice arrow laws
left (arr f) ≈ arr (left f)
left (f >>> g) ≈ left f >>> left g
f >>> arr Left ≈ arr Left >>> left f
left f >>> arr (id +++ g) ≈ arr (id +++ g) >>> left f
left (left f) >>> arr assocsum ≈ arr assocsum >>> left f

-- Definition of [assoc] (used in the last arrow law)
assoc ((a,b),c) = (a,(b,c))
unassoc (a,(b,c)) = ((a,b),c)
```

```
-- Definition of [assocsum] (used in the last choice arrow law)
assocsum (Left (Left x)) = Left x
assocsum (Left (Right y)) = Right (Left y)
assocsum (Right z) = Right (Right z)

unassocsum (Left x) = Left $ Left x
unassocsum (Right (Left y)) = Left $ Right y
unassocsum (Right (Right z)) = Right z
```

B APPENDIX: ROCQ PROVER FORMALIZATION OF FREER ARROW EQUIVALENCE

```
Fixpoint CharacteristicType {E : Type -> Type -> Type} {X Y : Type}
                            (e : FreerArrow E X Y) : Type :=
  match e with
  | Hom _ => Y
  | @Comp _ _ _ A B C f e y => A * (B -> CharacteristicType y)
  end.

Fixpoint character {E : Type -> Type -> Type} {X Y : Type}
  (e : FreerArrow E X Y) : X -> CharacteristicType e :=
  match e with
  | Hom f => f
  | Comp f _ y => join f (character y)
  end.

Inductive ArrowSimilar {E X Y P} :
  FreerArrow E X Y -> FreerArrow E P Y -> Prop :=
| HomSimilar : forall f g, ArrowSimilar (Hom f) (Hom g)
| CompSimilar : forall A B C D x y
                  (f : X -> A * C) (g : P -> A * D) (e : E A B),
    ArrowSimilar x y ->
    ArrowSimilar (Comp f e x) (Comp g e y).

Theorem ArrowSimilarCharTypEq {E X Y P} :
  forall (x : FreerArrow E X Y) (y : FreerArrow E P Y),
    ArrowSimilar x y ->
    CharacteristicType x = CharacteristicType y.
Proof. (* Omitted *) Qed.

Variant ArrowEq {E X Y} : FreerArrow E X Y -> FreerArrow E X Y -> Prop :=
| ArrowEqSimilar : forall x y (H : ArrowSimilar x y),
    (** This is essentially [character x = character y]. We need this stated in
    this awkward way to convince Rocq that [character x] and [character y] share
    the same type, so we can use equality on them. *)
    (let H0 := ArrowSimilarCharTypEq x y H in
     let cx := eq_rect _ (fun T : Type => X -> T) (character x) _ H0 in
     cx = character y) ->
    x ≈ y
```

```
where "x ≈ y" := (ArrowEq x y).
```

C APPENDIX: COMPOSABLE EFFECTS

```
-- |- The [Sum2] datatype.
data Sum2 (l :: Type -> Type -> Type) (r :: Type -> Type -> Type) a b where
  Inl2 :: l a b -> Sum2 l r a b
  Inr2 :: r a b -> Sum2 l r a b

-- |- An injection relation between effects.
class Inj2 (a :: Type -> Type -> Type) (b :: Type -> Type -> Type) where
  inj2 :: a x y -> b x y

-- |- Automatic inference using typeclass resolution.
instance Inj2 l l where
  inj2 = id

instance Inj2 l (Sum2 l r) where
  inj2 = Inl2

instance Inj2 r r' => Inj2 r (Sum2 l r') where
  inj2 = Inr2 . inj2

-- |- A more generalized instance showing that freer arrows are an ArrowState.
instance Inj2 (StateEff s) e => ArrowState s (FreerArrow e) where
  get = embed $ inj2 Get
  put = embed $ inj2 Put
```

D APPENDIX: SIMPLE OPTIMIZATIONS BASED ON BRIDGED FREER ARROWS

```
getget :: FreerArrow (IndexedMapStateEff k v) a b ->
          FreerArrow (IndexedMapStateEff k v) a b
getget (Comp _ (GetIM _) (Comp IdBridge (GetIM k2) c)) = Comp IdBridge (GetIM k2) c
getget (Comp f e c) = Comp f e $ getget c
getget x = x

getput :: Eq k => FreerArrow (IndexedMapStateEff k v) a b ->
                  FreerArrow (IndexedMapStateEff k v) a b
getput (Comp _ (GetIM k) (Comp IdBridge (PutIM k') c))
  | k == k' = Comp IdBridge (GetIM k) c
getput (Comp f e c) = Comp f e $ getput c
getput x = x

putget :: Eq k => FreerArrow (IndexedMapStateEff k v) a b ->
                  FreerArrow (IndexedMapStateEff k v) a b
-- Note the use of IdBridge here; we can't have an arbitrary bridge on the left
putget (Comp IdBridge (PutIM k) (Comp IdBridge (GetIM k') c))
  | k == k' = Comp IdBridge (PutIM k) c
```

```
putget (Comp f e c) = Comp f e $ putget c
putget x = x

putput :: Eq k => FreerArrow (IndexedMapStateEff k v) a b ->
          FreerArrow (IndexedMapStateEff k v) a b
putput (Comp f (PutIM k1) (Comp IdBridge (PutIM k2) c))
  | k1 == k2  = Comp f (PutIM k1) c
  -- otherwise, fall through; writes to different keys still need to happen
putput (Comp f e c) = Comp f e $ putput c
putput x = x
```