Freer Arrows and Why You Need Them in Haskell

Grant VanDomelen
gsv@pdx.edu
Portland State University
USA

Gan Shen
gshen42@ucsc.edu
University of California, Santa Cruz
USA

Lindsey Kuper
lkuper@ucsc.edu
University of California, Santa Cruz
USA

Yao Li
liyao@pdx.edu
Portland State University
USA

Abstract

Freer monads are a useful structure commonly used in various domains due to their expressiveness. However, a known issue with freer monads is that they are not amenable to static analysis. This paper explores freer arrows, a relatively expressive structure that is amenable to static analysis. We propose several variants of freer arrows. We conduct a case study on choreographic programming to demonstrate the usefulness of freer arrows in Haskell.

ACM Reference Format:
Grant VanDomelen, Gan Shen, Lindsey Kuper, and Yao Li. 2025. Freer Arrows and Why You Need Them in Haskell. In *Proceedings of the ACM SIGPLAN Haskell Symposium 2025 (Haskell 2025).* ACM, New York, NY, USA, 16 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 [Koh et al. 2019; Letan et al. 2018; Ye et al. 2022; Zhang et al. 2021a], embeddings [Chlipala 2021; Christiansen et al. 2019; Korkut et al. 2025], information-flow analysis [Silver et al. 2023; Silver and Zdancewic 2021], testing [Li et al. 2021], *etc.*

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

Haskell 2025, Oct. 16–17, Singapore

Motivating example. To illustrate the problem, let's imagine that we are building an embedded domain-specific language (EDSL) for web services in Haskell. We can either get from a server or post to a server in this EDSL. The implementation of such get and post operations can be different depending on the protocols and the web service, so we want to have an abstract interface for these operations.[1]

We can implement this interface using freer monads. There are many variants of free and freer monad definitions, but, for simplicity, let's use the following freer monads:

```
data FreerMonad (e :: Type -> Type) (a :: Type) where
  Ret :: a -> FreerMonad e a
  Eff :: e x -> (x -> FreerMonad e a) ->
         FreerMonad e a
```

A freer monad is an algebraic data type parameterized by an "effect" `e` and a return type `a`. It has two data constructors: a `Ret` constructor for pure computations; and an `Eff` constructor for effects and a continuation that reacts to the result of an effect.

In our web service example, the effect can be defined as the following generalized algebraic datatype (GADT):

```
data WebServiceOps :: Type -> Type where
  Get  :: URL -> [String] -> WebServiceOps String
  Post :: URL -> [String] -> String -> WebServiceOps ()
```

Such an interface enables us to define the get and post operations as the following functions:

```
get :: URL -> [String] ->
      FreerMonad WebServiceOps String
get url params = Eff (Get url params) Ret

post :: URL -> [String] -> String ->
      FreerMonad WebServiceOps String
post url params body = Eff (Post url params body) Ret
```

These functions simply "embed" an "abstract syntax node" representing a get or a post operation in a freer monad. To implement the underlying operations, we need to implement *effect handlers*. For example, one possible effect handler would have the following type signature:

[1]The example is adapted from a similar example shown in Capriotti and Kaposi [2014, Section 1.3].

```
handleWebServiceOps :: WebServiceOps a -> IO a
```

But the user can choose to implement such a handler using other monads as well. For example, a user can implement a handler with a state monad to run simulations without actually invoking I/O.

A key advantage of freer monads is that they can be interpreted into *any monads*, as long as there is an effect handler for such monad. More concretely, we can define the following interpretation function for freer monads:

```
interp :: Monad m => (forall a. e a -> m a) ->
                     FreerMonad e a -> m a
interp _      (Ret a)     = return a
interp handle (Eff eff k) =
  handle eff >>= interp handle . k
```

Challenge: static analysis. So far so good. But what if we want to do some static analysis on our EDSL? Perhaps we want to count the number of get and post operations, respectively. Or perhaps we want to collect on the URLs that we are posting to for security checks. Since we are implementing an EDSL, we would like to implement such a static analyzer *as a function* inside Haskell as well.

Unfortunately, we cannot do that with this approach based on freer monads. This is because monads are so expressive, that we can easily define the following function:

```
postDepends :: URL -> [String] -> String ->
               FreerMonad WebServiceOps ()
postDepends url params body =
  get url params >>=
  postNTimes url params body .
  (read :: String -> Integer)
```

Where `postNTimes url params body n` posts to `url` with parameters `params` and content body `body` for `n` times. In this `postDepends` function, the number of post operations cannot be counted, as the number depends on the result of the first get operation. We can only know the result of a get operation *after* we actually perform the get operation.

However, it is unsatisfying that we cannot statically analyze the following code, either:

```
echo :: URL -> URL -> [String] ->
        FreerMonad WebServiceOps ()
echo url1 url2 params =
  get url1 params >>= post url2 params
```

In this case, only the content body of the post operation depends on the result of the get operation. If we look at the code, we know that there must be an equal amount of get and post operations. Furthermore, if none of `url1` or `url2` depend on an effectful operation, we should be able to collect all the URLs we are getting from or posting to.

Sadly, we cannot write a function to statically analyze this piece of code, either, due to the expressiveness of monads. Indeed, this dynamic nature of monads has been discussed in literatures under various contexts [Capriotti and Kaposi 2014; Li and Weirich 2022; Mokhov et al. 2019, 2020].

Our work. 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. In this paper, we propose several types of *freer arrows* and explore their usefulness in Haskell.

Prior works like Rivas and Jaskelioff [2017] have explored the concept of free arrows, a datatype replying on *profunctors*. Our definitions are *freer* because we remove the dependency on profunctors, following the distinction set by Kiselyov and Ishii [2015].

We make the following contributions:

- We define three basic types of freer arrows, including *freer pre-arrows* (Section 3), *freer arrows* (Section 4), and *freer choice arrows* (Section 5).
- We present a case study with freer arrows on *choreographic programming* (CP) based on the HasChor framework [Shen et al. 2023]. We show that are able to implement *static endpoint projection*, *endpoint static analysis*, and *selective broadcasting* with HasChor implemented using freer arrows (Section 6).

In addition, we provide an overview of our approach in Section 2, we discuss other aspects of freer arrows, related work, and future work in Sections 7 and 8. We conclude this paper in Section 9.

2 Freer Arrows: An Overview

We show a slightly modified version of Hughes [2000]'s arrows in the top half of Fig. 1. An arrow is first a `Category`. A `Category` has two methods: `id` for identity category, and `(.)` for composing two categories whose input and output match. We also define a commonly used operator `(>>>)` for categories, which essentially flips the arguments of `(.)`. A `PreArrow` adds to `Category` an additional `arr` method that "converts" a function to a category. An `Arrow` adds an additional `first` method to a `PreArrow` that allows it to carry extra values using a product in both its input and output.

Compared with typeclasses like functors/applicative functors/monads, arrows additionally contain the input type as part of its type.

Functions are a classic example of arrows. Every monad can also be made into an arrow by adding the input to its type [Hughes 2000]. For example, state is an instance of arrows, as demonstrated in the bottom half of Fig. 1.

The freer arrow datatype. The key idea of our approach is based on the freer arrow datatype. Given any effect datatype `e` of kind `Type -> Type -> Type`, applying the freer arrow datatype to `e` will result in an instance of `Arrow`. Here is one

```haskell
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

-- 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 (.)

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)

-- |- The state arrow.
newtype AState s a b =
  AState { runAState :: (a, s) -> (b, s) }

instance Category (AState s) where
  id = AState id
  AState f . AState g = AState $ f . g

instance PreArrow (AState s) where
  arr f = AState $ first f

instance Arrow (AState s) where
  first  (AState f) =
    AState $ \((a, c), s) ->
    let (b, s') = f (a, s) in ((b, c), s')
```

Figure 1. Key typeclass definitions related the arrows and state (*i.e.,* `AState`) as an instance of arrows. We defined `AState` with state `s`, input `a`, and output `b`. The function is intentionally uncurried, so we have simpler definition for methods like `id`, `(.)`, and `arr`. Note that we define these methods by using the `id`, `(.)`, and `first` methods of function arrows, respectively.

definition of the freer arrow datatype (or simply *freer arrows* for short in the rest of this paper):

```haskell
data FreerArrow e x y where
  Hom :: (x -> y) -> FreerArrow e x y
  Comp :: (x -> (a, c)) -> e a b ->
          FreerArrow e (b, c) y -> FreerArrow e x y
```

Freer arrows have two data constructors: `Hom` and `Comp`. The `Hom` constructor "embeds" pure computations in a freer arrow. The `Comp` constructor, on the other hand, "embeds" an effect `e` that has an input `a` and output `b`. Together with the effect `e a b`, the `Comp` constructor also contains (1) a function for retrofiting an input `x` to the effect's input `a` with a carried value of type `c`, and (2) a continuation freer arrow that takes the effect's output `b` as an input with the previously carried value of type `c`. The carried value of type `c` may look bizarre at this point, but it is crucial to *strengthening* freer arrows for implementing the `first` method of arrows. We defer the detailed definitions of these methods to Sections 3 and 4.

The web service example, revisited. To use freer arrows for our web service example, we can encode the web service interface using the following effect datatype:

```haskell
data WebServiceOps :: Type -> Type -> Type where
  Get  :: URL -> [String] -> WebServiceOps () String
  Post :: URL -> [String] -> WebServiceOps String ()
```

Compared with the `WebServiceOps` shown in Section 1, this datatype additionally contains the type of the input as its first type argument.

We can further define the following "smart constructors" for `Get` and `Post`:

```haskell
embed :: e x y -> FreerArrow e x y
embed f = Comp (,()) f (arr fst)

get :: URL -> [String] ->
       FreerArrow WebServiceOps () String
get url params = embed $ Get url params

post :: URL -> [String] ->
        FreerArrow WebServiceOps String ()
post url params = embed $ Post url params
```

The `embed` function "embeds" an effect in a `Comp`. `Comp` requires an existential type, *i.e.,* the type of the carried value, but we do not need to carry any values in this case, so we just use `()` as that type. Then `get` and `put` are defined as embedding `Get` and `Put`, respectively.

We can now re-implement the `echo` function shown previously using freer arrows:

```haskell
echo :: URL -> URL -> [String] ->
        FreerArrow WebServiceOps () ()
echo url1 url2 params =
  get url1 params >>> post url2 params
```

Syntactically, this definition is almost the same as our previous definition of `echo`, except that we replace `>>=` with `>>>`. However, because this new function is defined using freer arrows, we *can* do static analysis on it now. For example, the following function counts the number of effects in any programs implemented using freer arrows:

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

Just like freer monads, we can implement a freer arrow into any arrows, as long as we can provide an effect handler:

```
interp :: Arrow arr =>
  (e :-> arr) -> FreerArrow e x y -> arr x y
interp _       (Hom f) = arr f
interp handler (Comp f x y) =
  arr f >>> first (handler x) >>> interp handler y
```

Where the type operator :-> is defined as follows:[2]

```
type p :-> q = forall a b. p a b -> q a b
```

Why static analysis matters. It might seem that we are giving up the expressiveness of monads for the niche advantage of counting the number of effects. However, the significance of being able to define a function to statically analyze a program should not be underestimated.

To demonstrate the usefulness of static analysis in pratice, we will show some applications in *choreographic programming* (CP). In choreographic programming, we write one program that will be *projected* to multiple *endpoints* that are clients or servers participating in the communications. Shen et al. [2023] showed that CP can be implemented as an EDSL in Haskell using freer monads as a library called HasChor.

In Section 6, we re-implemented HasChor using freer arrows. We are able to implement the following features that were not possible in the original HasChor:

- *Static endpoint projection and endpoint static analysis.* We can perform *endpoint projection* statically, so that each endpoint only has their "full program" before execution. These "full programs" themselves are statically analyzeable, so we can inspect information such as if an endpoint is only communicating with trusted servers, or which endpoints are intensive on local computation and which endpoints heavy on network communications, *etc.*
- *Selective broadcasting*. When a choice needs to be synchronized among multiple parties, we can statically analyze the choreography to find all the involved participants. In this way, we can project the choreography such that only involved parties will be targets of a broadcast. This allows a lightweight implementation of mini-*conclaves* [Bates et al. 2024].

What about conditionals or loops? But would we be limiting the expressiveness too much by using freer arrows? What if we need conditionals or loops in our EDSL?

Implementing conditionals is straightforwad if we use a more expressive variant of freer arrows: freer choice arrows. Freer choice arrows implement the `ArrowChoice` typeclass that allows branching based on a sum type. We defer more detailed discussion of `ArrowChoice` and freer choice arrows to Section 5.

For loops, we consider two situations: when the number of iterations is statically known, and when it's not. If the number of iterations is statically known, we can implement such a loop by a recursion based on the number of iterations. If the number of iterations is not statically known, we need to use additional datatypes for *reifying* loops combined with freer choice arrows for modeling conditionals. We briefly discuss one such an example datatype in Section 7 and Appendix D. Notably, by using a datatype for loops, we lose the ability to statically analyze the entire program. However, we are still able to statically analyze the loop body or the part before/after the loop, if they are defined using freer arrows.

What about other freer datatypes? Though we have demonstrated that freer arrows can help us with this web service example, a reader might still (rightfully) wonder: Why do we choose freer arrows? Why not use other freer datatypes like freer applicative functors [Capriotti and Kaposi 2014] or freer selective functors [Mokhov et al. 2019]? Existing works have shown that we can define functions that perform static analysis on these datatypes as well [Capriotti and Kaposi 2014; Li and Weirich 2022; Mokhov et al. 2019].

It turns out that arrows have just the right expressiveness for our purpose, a fact that was first demonstrated by Lindley et al. [2008]. In particular, they show that applicative functors are as expressive as any arrows `a` for which there is an isomorphism between `a b c` and `a () (b -> c)`. This isomorphism indicates that applicative functors are essentially arrows whose input type can always be considered as `()`, *i.e.,* an effectful computation never uses the result of another effectful computation as input in applicative functors. This means that we *cannot* even implement the `echo` function if we use a freer applicative functor!

3 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` of kind `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 3–5). 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).

To show that a freer pre-arrow is a `Category`, we need to define both the **`id`** and `(.)` methods (lines 7–13). The **`id`** method can be defined as `Hom id`, where the **`id`** in the definition is the identity function (line 9). Composition `(.)` is defined by pattern matching on both arguments. When both

[2] We borrow this notation from the `profunctors` library by Edward Kmett: http://github.com/ekmett/profunctors/.

```
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 ->
5              FreerPreArrow e x y
6
7  -- | Freer pre-arrows are categories.
8  instance Category (FreerPreArrow e) where
9    id = Hom id
10
11   Hom f        . Hom g        = Hom (f . g)
12   Comp f' e y . Hom g        = Comp (f' . g) e y
13   f            . Comp f' e y = Comp f' e (f . y)
14
15 -- | Freer pre-arrows are pre-arrows.
16 instance PreArrow (FreerPreArrow e) where
17   arr = Hom
18
19 -- | Embed an effect in freer arrows.
20 embed :: e x y -> FreerPreArrow e x y
21 embed f = Comp id f id
22
23 -- | The type for effect handlers.
24 type x :-> y = forall a b. x a b -> y a b
25
26 -- | Freer pre-arrows can be interpreted into any
27 -- pre-arrows, as long as we can provide an effect
28 -- handler.
29 interp :: Arrow arr =>
30   (e :-> arr) -> FreerPreArrow e x y -> arr x y
31 interp _       (Hom f)       = arr f
32 interp handler (Comp f x y)  =
33   arr f >>> (handler x) >>> interp handler y
```

Figure 2. Key definitions of freer pre-arrows (`FreerPreArrow`) in Haskell. The code is simplified for presentation. In our artefact, we additionally show that `FreerPreArrow` is a `Profunctor`, we then use the `lmap` method of `Profunctors` to define `(.)`. The definition shown here is equivalent to the definition based on `Profunctors`.

arguments are `Homs`, the definition is just applying `Hom` to the function composition (line 11). When the first argument is `Comp f' x y` and the second argument is `Hom g`, we simply compose functions `f'` with `g` (line 12). When the second argument is a `Comp f' x y`, we make a recursive call to compose the first argument with `y` (line 13). Note that even though we use `(.)` in the definition in every case, only the last one on line 13 is a recursive call; all other `(.)` are function compositions.

Freer pre-arrows are also `PreArrow` (lines 15–17). The `arr` method is simply `Hom` (line 17).

Given an effect `e x y`, we can `embed` the effect in a freer pre-arrow (lines 19–21). We embed such an effect using the `Comp` constructor with an identity function (the first **`id`** on line 21) and an identity freer pre-arrow (the second **`id`** on line 21).

Finally, we can interpret a `FreerPreArrow` to any pre-arrow if we provide an "effect handler" (lines 23–33). 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 input type `a` and output type `b` (lines 23–24). When interpreting a `FreerPreArrow`, we do a case analysis on the freer pre-arrow. In the case of `Hom f`, we simply apply the `arr` method of the pre-arrow `arr` to `f` (line 31). 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` (lines 32–33).

Expressiveness. Freer pre-arrows are sufficient for defining functions like `echo` in our web service example. Indeed, we can change the type signature of `get`, `post`, and `echo` functions defined in the previous section to use `FreerPreArrow` instead.

However, freer pre-arrows only allows the use of effect in a streamlined manner. For example, we cannot "share" the output of an effect as input of multiple other effects. To do so, we need the `first` method defined on `Arrows` (also known as the *strength* of arrows). We talk about freer arrows that allow this capability in Section 4.

Static analysis. We can statically analyze a freer pre-arrow to obtain its "approximation". For example, the `count` function defined in the previous function also works on freer pre-arrows. In fact, we can define the following generalized `approximate` function:

```
approximate :: Monoid m =>
               (forall x y. e x y -> m) ->
               FreerPreArrow e a b -> m
approximate _ (Hom _) = mempty
approximate f (Comp _ e y) = f e <> approximate f y
```

Given an "approximation function" `f` that approximates an effect to a monoid `m`, we can iterate over the entire freer pre-arrow and collect all the effect approximations via `(<>)`.[3] For example, the `count` function is a special case of the `approximate` function:

```
count :: FreerPreArrow e x y -> Int
count = getSum . approximate (const $ Sum 1)
```

Here we use the `Sum` type from Haskell's `Data.Monoid` module. Recall that in Haskell, **`Int`**s are not `Monoids`. We need to specify whether the monoid operations correspond to plus or multiplication using the `Sum` type or `Product` type.

[3] The function does essentially the same thing as `foldMap`, but it cannot be defined as a `foldMap` as it has a different type signature.

```
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 ->
5            FreerArrow e x y
6
7  -- | Freer arrows are arrows.
8  instance Arrow (FreerArrow e) where
9    first :: FreerArrow e a b ->
10            FreerArrow e (a, c) (b, c)
11   first (Hom f) = Hom $ first f
12   first (Comp f a b) =
13     Comp (first f >>> assoc)
14          a
15          (lmap unassoc (first b))
16
17 -- | Embed an effect in freer arrows.
18 embed :: e x y -> FreerArrow e x y
19 embed f = Comp (,()) f (arr fst)
20
21 -- | Freer arrows can be interpreted into any
22 -- arrows, as long as we can provide an effect
23 -- handler.
24 interp :: Arrow arr =>
25   (e :-> arr) -> FreerArrow e x y -> arr x y
26 interp _       (Hom f) = arr f
27 interp handler (Comp f x y) =
28   arr f >>> (first (handler x)) >>>
29   interp handler y
30
31 -- Helper functions. Definitions omitted.
32 assoc :: ((a,b),c) -> (a,(b,c))
33 unassoc :: (a,(b,c)) -> ((a,b),c)
```

Figure 3. Key definitions of freer arrows (`FreerArrow`) in Haskell. Instances of `Category` and `PreArrow` are omitted as they are the same as those of `FreerPreArrows` shown in Fig. 2.

We can do more than `count` with `approximate`. For example, we can also collect a *trace* of all the effects that will happen in a freer pre-arrow. We defer more examples demonstrating usefulness of static analysis to Section 6.

4 Freer arrows

To enable "sharing" the result of an effect among multiple other effects instead of only the next effect, we need to enhance freer pre-arrows with *products* to enable defining `first`. This gives us freer arrows. It turns out that we only need to modify the `Comp` constructor to obtain freer arrows. We show the definition in Fig. 3.

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)` to allow a carried value of type `c` (line 3). Correspondingly, the input type of the inner freer arrow is changed to `(b, c)` (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).

`FreerArrow`s are instances of `Category`, and `PreArrow`. These definitions are the same as those of freer pre-arrows, so we omit them here.

To show that `FreerArrow`s are an instance of `Arrow`, we need to define the `first` method (racall Fig. 1). We show the type signature of `first` for freer arrows in lines 9–10. We define this method by pattern matching on `FreerArrow`s. In the case of `Hom f`, we apply the `first` method of the function arrow to function `f` (line 11). 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 `((_, _), _)` (line 13). To match the type with that of the effect `a`, we then call `assoc` to convert the product to the form of `(_, (_, _))` (line 13). Finally, we call `unassoc`, the inverse of `assoc` on the recursive evaluation `first b` (line 15). 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 `FreerArrow`s.

Given an effect, we can `emded` it in a freer arrow using the `Comp` constructor (lines 17–19). The definition here is more complicated than that of freer pre-arrows, because we need to apply `Comp` to a function argument and a continuation arrow that contain a carried value. However, we do not need any carried value at this point, so we can simply use `()`. Our function argument wraps the input in a pair whose second element is `()`; our continuation arrow uses `arr fst` to drop the `()` from the pair.

We can interpret a freer arrow to any arrows using the `interp` function (lines 21–29). Compared with the `interp` function of `FreerPreArrow`, we need to call `first` on `handler x`, due to the type change in `f` (line 28). This is inevitable even for the segments of a freer arrow that we do not use `first`.

Expressiveness. Based on the `first` method, we can further define some other useful combinators shown in Fig. 4. Thanks to these combinators, we can now "share" the result of an effect thanks to these methods. Taking the web service example again, we can define the following function that gets the content from `url1`, and then forward it to both `url2` and `url3`:

```
forward :: URL -> URL -> URL -> [String] ->
           FreerArrow WebServiceOps () ()
forward url1 url2 url3 params =
  get url1 params >>>
  post url2 params &&& post url3 params >>>
  arr (const ())
```

```haskell
(***) :: Arrow a => a b c -> a d e -> a (b, d) (c, e)
(&&&) :: Arrow a => a b c -> a b c' -> a b (c,c')
second :: Arrow a => a b c -> a (d,b) (d,c)
```

Figure 4. Other useful combinators of `Arrow` that can be defined using the `first` method [Hughes 2000]. In Haskell, these functions are also defined as methods of `Arrows`. An `Arrow` instance can be defined by either `first` or `(***)`.

Even though we now have the ability to "pass down" values, freer arrows are *very static*. We cannot write a program that has conditionally executed branches. This means that every effect embedded in a freer arrow *will* happen, a property that may not always be desirable. Fortunately, we can further enhance freer arrows to allow conditionals and branching. We will talk about freer choice arrows, the datatype that allows these features in Section 5.

Static analysis. Freer arrows can be statically analyzed in the same way as freer pre-arrows. In fact, the `approximate` function defined in Section 3 works for freer arrows if we simply modify its type signature.

Why are freer arrows defined in this way? Our freer pre-arrow and freer arrow definitions are based on the free arrows of Rivas and Jaskelioff [2017]. However, Rivas and Jaskelioff's definition is based on another concept known as *profunctors*. Their free arrow is a pre-arrow only if the effect datatype `e` is a profunctor and it is an arrow only if `e` is a *strong profunctor*. Our definition is *freer*—a notation coined by Kiselyov and Ishii [2015]—because we do not have such requirements on `e`. Our key observation is that we can inline a free profunctor inside the free arrow of Rivas and Jaskelioff. After that, we only need some simplifications to obtain `FreerPreArrows`. Similarly, we can inline free strong profunctors to obtain `FreerArrows`. We describe this process in more detail in Appendix A.

Another way of defining freer pre-arrows and freer arrows is *reifying* all the operators of pre-arrows and arrows, similar to the reified datatypes studied by Li and Weirich [2022]. However, such an approach would result in a datatype with more constructors, which brings additional trouble to static analysis because we need to match all constructors. Our definition, in comparison, represents a *normalized* freer arrow.

Finally, we can define freer pre-arrows and freer arrows using final encodings, *i.e.*, defining them by their interpreters. For example, freer arrows can be defined as follows:

```haskell
newtype FreerArrowFinal e b c = FreerArrowFinal {
  runFreer :: forall a. Arrow a =>
              (e :-> a) -> a b c }
```

We choose not to use this version because it does not have any data constructors, which makes it difficult to perform any static analysis that cannot be defined by `runFreer`. If we want, we can always interpret our freer arrows to this `FreerArrowFinal`.

```haskell
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 Arrow (FreerChoiceArrow e) where
9    first (Hom f) = Hom $ first f
10   first (Comp f a b) =
11     Comp (first f >>> distr >>> left assoc)
12          a
13          (lmap (left unassoc >>> undistr)
14                (first b))
15
16 -- | The ArrowChoice typeclass.
17 class Arrow a => ArrowChoice a where
18   left :: a b c -> a (Either b d) (Either c d)
19
20 -- | Freer choice arrows are an instance of
21 -- [ArrowChoice].
22 instance ArrowChoice (FreerChoiceArrow e) where
23   left (Hom f) = Hom $ left f
24   left (Comp f a b) =
25     Comp (left f >>> assocsum)
26          a
27          (lmap unassocsum (left b))
28
29 -- Helpful functions. Definitions omitted.
30 distr   :: (Either (a, b) c, d) ->
31            Either ((a, b), d) (c, d)
32 undistr :: Either ((a, b), d) (c, d) ->
33            (Either (a, b) c, d)
34 assocsum   :: Either (Either x y) z ->
35               Either x (Either y z)
36 unassocsum :: Either x (Either y z) ->
37               Either (Either x y) z
```

Figure 5. Key definitions for `FreerChoiceArrow`. Instances of `Profunctor`, `Category`, and `PreArrow` are omitted as they are the same as those of `FreerPreArrows` shown in Fig. 2.

5 Freer Choice Arrows

When conditionals and branching are required, we need freer choice arrows. We show the key definitions of freer choice arrows in Fig. 5.

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)`,

```
embed :: e x y -> FreerChoiceArrow e x y
embed f = Comp (Left . (,())) f (arr (fst ||| id))

interp :: (Profunctor arr, ArrowChoice arr) =>
          (e :-> arr) ->
          FreerChoiceArrow e x y -> arr x y
interp _       (Hom f) = arr f
interp handler (Comp f x y) =
  lmap f (left (first (handler x))) >>>
  interp handler y
```

Figure 6. The `embed` and `interp` methods of freer choice arrows.

```
(+++) :: ArrowChoice a =>
         a b c -> a b' c' ->
         a (Either b b') (Either c c')
(|||) :: ArrowChoice a =>
         a b d -> a c d -> a (Either b c) d
right :: ArrowChoice a =>
         a b c -> a (Either d b) (Either d c)
```

Figure 7. Other useful combinators of `ArrowChoice` that can be defined using the `left` method [Hughes 2000]. In Haskell, these functions are also defined as methods of `ArrowChoice`s. An `ArrowChoice` instance can be defined by either `left` or `(+++)`.

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—representing a branch that the effect does not happen. Finally, the inner `FreerChoiceArrow` needs to handle the new input type `Either (b, c) w` (line 5).

We also show the definitions of `first` method in lines 8–14. In the `Hom` case, we simply use the `first` method of function arrows (line 9). The `Comp` case is more complex, but the key idea is using helper functions shown in lines 30–37 to shuffle around the values to match the types. These functions do not pose computational significance.

In addition, we show that freer choice arrows are an instance of `ArrowChoice`. We show the definition of `ArrowChoice` in lines 16–18. The key method of `ArrowChoice` is `left`, which branches on a sum type `Either b d`. In the left case, it runs the arrow `a b c` and returns as a left injection of `Either c d`. In the right case, it does nothing and simply returns the value of type `d` as a right injection of `Either c d`. The definition of `left` method on freer choice arrow is shown in lines 20–27. Like the `first` method, most of the code on lines 25–27 is to shuffle around values to match the types.

We define the `embed` and `interp` functions of freer choice arrows in Fig. 6. The `embed` function wraps an effect inside the `Comp` constructor. Given any input, we first combine it with `()` to make a pair, then make the pair a left injection of a sum type. This is encoded by the function argument `Left . (,())`. After executing the effect, we pattern match on the result of the arrow. If we are on the left branch, we discard the second element in the pair; if we are on the right branch, we do nothing. This is encoded by the continuation arrow `arr (fst ||| id)`. We will explain what `(|||)` operator means shortly in this section.

The `interp` function is similar to that of freer arrows, except that, in the `Comp` case, we need to apply the `left` method after applying `first` to the result of `handler` application.

Expressiveness. Based on the `left` method, we can further define some other useful combinators shown in Fig. 7. If we replace the freer arrows used in the web service example with freer choice arrows, we can then implement programs that send messages to different `URL`s depending on a result. For example:

```
forwardIf :: URL -> URL -> URL -> [String] ->
             String -> String ->
             FreerChoiceArrow WebServiceOps () ()
forwardIf url1 url2 url3 params m1 m2 =
  get url1 params >>> arr (read :: String -> Int) >>>
  arr (\n -> if n > 0 then Left m1 else Right m2) >>>
  post url2 params ||| post url3 params
```

Here, we replace all the `get` and `post` functions with the version using freer choice arrows. The program first `get`s a message from `url1` and parses it as an `Int`. If the number if greater than `0`, we `post` to `url2` with message `m1`; otherwise, we `post` to `url3` with message `m2`.

Note that freer choice arrows only enable *finite* branching. We still cannot define functions like `postDepends` shown in Section 1, as that function requires *infinite* branching, *i.e.,* branching on all unbounded integers.

Static analysis. Branching in freer choice arrows allows us to skip any effects, so we can no longer accurately count the number of effects in a freer choice arrow. However, we can still approximate a freer choice arrow, except that such an approximation is an *over-approximation.* Such a function is in fact the same as the `approximate` function shown in Section 3. This is no coincidence, as we calculate the over-approximation by assuming all effects will happen.

We can also get an under-approximation of any freer choice arrows, but that would not be interesting as the under-approximation will always be `0`, *i.e.,* we skip all the effects.

6 Case Study: HasChor

In this section, we demonstrate the usefulness of freer arrows via a case study on HasChor, an EDSL for *choreographic programming* (CP) in Haskell [Shen et al. 2023].

CP is a programming paradigm where one writes a single program, a *choreography*, that describes the entire behavior of a distributed system. A separate process called *endpoint projection* compiles a choreography to correct-by-construction individual programs for each node.

The key idea of HasChor is that we can represent the program as a freer monad that encodes the communications among multiple parties. Thanks to freer monads, a user can re-use all existing advanced features from Haskell in HasChor, including higher-order functions, type systems, *etc.* For example, we can write a choreography for our previous echo example as follows:

```
echo :: Choreo IO (String @ "client")
echo = do
  strAtClient <- client `locally` (const getInput)
  strAtServer <- (client, strAtClient) ~> server
  (server, strAtServer) ~> client
```

The echo function has type `Choreo IO (String @ "client")`. It means that it is a choreography that uses `IO` for all local computations and returns a `String` at a location known as `"client"`. The type operation `a @ l` is a customized definition in HasChor that represent a value of type `a` at location `l`.

In this choreography echo, the client first tries to get an input string from users `locally` and bind it to `strAtClient`. It then sends the string to the server via the `(~>)` operator. The server receives the string as `strAtServer`. Finally, the server echoes the string back to the client using the `(~>)` operator again. The `locally` and `(~>)` operators have the following types:

```
locally :: Proxy l -> (Unwrap l -> m a) ->
           Choreo m (a @ l)
(~>)    :: (Proxy l, a @ l) -> Proxy l' ->
           Choreo m (a @ l')
```

Here we use Haskell's *phantom parameter* `Proxy`[4] to inform Haskell's type checker which location we are at. The `locally` function unwraps a local value at location `l` and executes it in a local monad `m` (`IO` in the case of echo). The `(~>)` operator sends a value of type `a` at location `l` to another location `l'`. This results in choreography with that value at location `l'`.

Note that how this choreography "takes no side". It describes what happens from a global view.

We can then *project* this choreography to each node—in this case, the client and the server—by interpreting all the multi-party communication events to *events specific to this endpoint*. This step is known as the endpoint projection. On the server side, we get a similar program to the echo function described in Section 1. On the client's side, we get a program that first tries to get an input from the user, then sends it to the server, and finally waiting to receive the echo from the server.

How is the endpoint projection implemented? Well, by interpreting freer monads! Indeed, the definition of endpoint projection in HasChor is essentially as follows:

```
epp :: Choreo m b a -> LocTm -> Network m b a
epp c l = interp handle c
  where handle = ... -- Definition omitted.
```

Both `Choreo` and `Network` in the type signature are freer monads. We show the definitions of `Choreo` and `Network` in Fig. 8. The `Choreo` type contains two effect events: `Local`, which represents local computation at a location, and `Comm`, which represents a communication between two locations. The `Network` type contains three effect events: `Run`, a local computation; `Send`, which sends a message to another location; and `Recv`, which waits to receive a message from another location. For now, let's ignore `Cond` in `ChoreoSig` and `BCast` in `NetworkSig`. The local function `handle` "translates" events in `ChoreoSig` to events in `NetworkSig`. Such "translation" depends on `l`, the second argument of `epp`, which represents which endpoint we are currently projecting to. For example, projecting `(client, strAtClient) ~> server` on the `client` results in a `Send` event; but projecting it on the `server` results in a `Recv` event.

Dynamic vs. static endpoint projection. One issue with HasChor is that the endpoint projection is too *dynamic* due to freer monads. To understand why, let's revisit the `interp` function of freer monads:

```
interp :: Monad m => (forall a. e a -> m a) ->
                     FreerMonad e a -> m a
interp _      (Ret a)     = return a
interp handle (Eff eff k) =
  handle eff >>= interp handle . k
```

The function pattern matches on the `FreerMonad` and applies `handle` when it's an `Eff` case. However, only the first `interp` is fully applied in this function. All recursive calls to `interp` are only partially applied! In fact, these recursive calls cannot be fully applied until we receive a result from the previous `handle eff`, and we can only get a result from a `handle eff` when we actually execute the program on a node. This means what each endpoint carries is not just their own program, but a "suspended" interpretation of the entire choreography. This is a known issue of HasChor. For example, Krook and Hammersberg [2024] try to solve this issue by using Template Haskell and compiler plugins.

We can solve this issue without metaprogramming by replacing freer monads with freer arrows. The change involves changing a number of types and replacing monadic operations with arrow operations, but most of these changes are straightforward. We show the new type definitions of `Choreo` and `Network` in Fig. 9. Interested readers can find a detailed

[4] https://hackage.haskell.org/package/base/docs/Data-Proxy.html

```
type Choreo m = FreerMonad (ChoreoSig m)

data ChoreoSig m a where
  Local :: KnownSymbol l => Proxy l ->
           (Unwrap l -> m a) ->
           ChoreoSig m (a @ l)
  Comm :: (Show a, Read a,
           KnownSymbol l, KnownSymbol l') =>
          Proxy l -> a @ l ->
          Proxy l' -> ChoreoSig m (a @ l')
  Cond :: (Show a, Read a, KnownSymbol l) =>
          Proxy l -> a @ l ->
          (a -> Choreo m b) -> ChoreoSig m b

type Network m = FreerMonad (NetworkSig m)

data NetworkSig m a where
  Run :: m a -> NetworkSig m a
  Send :: Show a => a -> LocTm -> NetworkSig m ()
  Recv :: Read a => LocTm -> NetworkSig m a
  BCast :: Show a => a -> NetworkSig m ()
```

Figure 8. The Choreo type and Network type in HasChor implemented using freer monads.

implementation of our work in our supplementary materials, which will be released as a publicly-available artifact.

With freer arrows, we can re-implement the previous echo choreography as follows:

```
echo = client `locally0` getInput >>>
       client ~> server >>>
       server ~> client
```

The new epp function also uses the interp method of freer arrows (Fig. 3) instead. Unlike the interp method of freer monads, recursive calls in interp of freer arrows are always fully applied.

Endpoint static analysis. What do we gain by having a static endpoint projection? For a start, we can now statically analyze every endpoint. For example, the following function collects all the effect events happened at an endpoint:

```
collect :: Network ar a b -> [Event]
collect = approximate (singleton . trace)
  where trace :: NetworkSig ar a b -> Event
        trace = ... -- Definition omitted.
```

The function works on the Network type and collects a list of effectful events of Local, Send, and Recv. Thanks to the static nature of freer arrows, we can fully apply the epp function to a program of Choreo type to obtain a program of Network type at an endpoint. We can then apply this collect function to statically analyze the program at an endpoint l.

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

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 b, Read b, KnownSymbol l) =>
          Proxy l -> Choreo ar b a ->
          ChoreoSig ar (b @ l) a

type Network ar = FreerArrow (NetworkSig ar)

data NetworkSig ar b a where
  Run :: ar b a -> NetworkSig ar b a
  Send :: Show a => LocTm -> NetworkSig ar a ()
  Recv :: Read a => LocTm -> NetworkSig ar () a
  -- [BCast] includes an extra argument to support
  -- selective broadcasting.
  BCast :: Show a => HashSet LocTm ->
           NetworkSig ar (a @ l) a
```

Figure 9. The Choreo type and Network type in HasChor implemented using freer arrows.

This can be useful for several reason. For example, we can check what other endpoints that l communicates with. If l is a secret endpoint that only communicates with other trusted endpoints, we can use this static analysis to ensure that all the endpoints l talk to are trusted. We can also check which endpoints are intensive on local computations and which endpoints are heavy on network communications. We can then use this information to for better resource allocation or endpoint placement.

Selective broadcasting. One feature in HasChor that we have not discussed so far is cond, a construct that implements the *knowledge of choice* [Carbone and Montesi 2013; Giallorenzo et al. 2024; Hirsch and Garg 2022].

We show an example choreography of a key-value server that uses cond in Fig. 10. The choreography contains a client, a primary server, and a backup server. The client first prepares and sends a request to the primary server. The primary server then handles the request locally. It then branches depending on whether the request is a Get or a Put. In the case of a Get, nothing special needs to be done. In the case of a Put, however, the primary server first forwards the request to the backup server, so that the backup server handles the request as well. Finally, no matter the result is a Get or a Put, the primary responds to the client with the result.

```
kvs :: Choreo IO (Response @ "client")
kvs = do
  req  <- client `locally` getRequest
  req' <- (client, req) ~> primary

  res  <- primary `locally` handleRequest req'
  cond (primary, req') \case
    Get k -> return ()
    Put k v -> do
      req'' <- (primary, req') ~> backup
      backup `locally` handleRequest req''
      return ()

  (primary, res) ~> client
```

Figure 10. An example key-value server with a client, a primary server, and a backup server implemented in HasChor.

The cond function is a thin wrapper of the Cond constructor in ChoreoSig in HasChor (Fig. 8). It takes a location l, a value at location l, and a continuation based on the value.

How do we deal with Cond in an endpoint projection? Because there might be additional participants involved in the continuation of Cond, *e.g.,* the backup server is involved in our kvs choreography, so we need to make sure all these involved participants synchronize, *i.e.,* they all enter the same conditional branch.

In HasChor, such a synchronization is implemented as a *broadcast*. Taking kvs as an example, the primary server will broadcast the request req' to all other endpoints. In endpoint projection, this means that Cond will be "translated" to the BCast event of NetworkSig (Fig. 8) to represent a broadcast. In the meantime, all other endpoints will try to Recv this message, so that they can enter the right branch depending on the same req'.

Ideally, this synchronization message should only be broadcast to endpoints that are involved. Unfortunately, finding out all involved parties requires a static analysis, which cannot be performed on freer monads! For this reason, HasChor chooses to broadcast the synchronization message to *all endpoints* instead. In the kvs example, this means that client is forced to participate in the synchronization, even though it is never mentioned in cond.

Such implementation is inefficient and scales poorly with more nodes. One recent work that tackles this issue is conclaves [Bates et al. 2024], which relies on dependent types.

However, *freer choice arrows* enable a more lightweight approach that does not require more complicated type systems. We show the Cond constructor of ChoreoSig in our version with freer arrows in Fig. 9.[5] Note that the continuation inside Cond is simply represented by a freer choice arrow, so it can be statically analyzed. Indeed, in our implementation of epp, we first use a static analysis to gather all the participants in this "continuation choreography". After that, we interpret Cond to a modified BCast event (Fig. 9) that carries an additional argument representing a set of locations. In this way, we only BCast to locations that are known to be involved due to static analysis. We show our implementation of kvs using freer arrows in Appendix B.

7 Discussion and Future Work

Composing effects. One advantage of freer monads or their variants is that we can compose effects to implement *algebraic effects*. Composing effects is fair game to freer arrows as well. We show an example of implementing effect composition in Appendix C.

Composing freer arrows. An alternative approach to building a hierarchy of freer arrows is *composable* freer arrows. For example, we can have all three variants of Comp constructors in freer pre-arrows, freer arrows, and freer choice arrows modeled separately, then allow users to compose them as needed. However, since the Comp constructors are recursive, we will run into the *expression problem* [Wadler 1998]. One way to work around this is using Church encodings [Delaware et al. 2013; Swierstra 2008]. Indeed, Li and Weirich [2022] have successfully applied this approach in the context of freer functors/applicative functors/monads. Overcoming the expression problem is out of the scope of this work, but interested readers can find more recent exploration on this topic in Kravchuk-Kirilyuk et al. [2024]; Rioux et al. [2023]; Zhang et al. [2021b], *etc.*

Loops. With freer arrows, we can only implement loops that have a fixed number of iterations. To implement other types of iterations, we need additional datatypes that reify those iterations. We show an example datatype that reifies the Elgot iteration [Adámek et al. 2011; Goncharov 2022] in Appendix D.

Optimizations. Static analysis usually helps with optimizations. In the future, we would like to look into optimizations based on freer arrows. A challenge is that the function arguments in Comp constructors are not amenable to static analysis. One way to deal with it is *defunctionalizing* common functions used for composing freer arrows, similar to Chupin and Nilsson [2019]. We have a prototype for these "defunctionalized" arrows but more experiments are required to implement useful optimizations.

[5] Freer choice arrows are not required for implementing constructors like Cond, but all the interesting cases with Cond only happen with branching (*e.g.,* the branching based on requests in kvs), so we need freer choice arrows to implement the case study.

Metaprogramming. In the future, we would like to explore combining freer arrows with metaprogramming. Since freer arrows enable static analysis, we should be able to exploit that to generate efficient code at compile time. Indeed, existing approaches have explored the use of applicative functors in metaprogramming [Gibbons et al. 2023; Willis et al. 2020]. We believe arrows should open up more opportunities. One challenge is again the function arguments in `Comp` constructors—we cannot simply "lift" them to quoted code in Template Haskell, but we might be able to do that if we limit the flexiblility of functions in arrows.

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.

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 structures, 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.

Choreographic Programming. The earliest incarnation of choreographies can be traced back to the mid-2000s in the context of web services [World Wide Web Consortium 2004, 2005, 2006]. At the time, choreographies were mainly used as *specifications* of communicating processes. Carbone and Montesi [2013]; Montesi [2013] pioneered choreographic *programming* languages with their Chor language and provided a correctness proof in terms of multiparty session types [Honda et al. 2008]. Since then, much work has focused on developing the theoretical foundations of CP, including its functional semantics [Cruz-Filipe and Montesi 2020; Hirsch and Garg 2022], location polymorphism [Graversen et al. 2024], and fully out-of-order execution [Plyukhin et al. 2024].

Implementations of CP started with Choral [Giallorenzo et al. 2024], a standalone language that extends Java with CP features. More recently, HasChor [Shen et al. 2023] popularized a library-level approach, providing an EDSL for writing choreographies, with endpoint projection realized as runtime dispatch. Since then, library-level CP have been applied to more languages, including Rust [Bates et al. 2024], Clojure [Lugović and Jongmans 2024], Elixir [Wiersdorf and Greenman 2024], and others.

Conditional execution is essential yet challenging in CP, as it must ensure knowledge of choice (KoC) [Castagna et al. 2011]—locations whose behavior varies across branches are properly informed of the selected branch/choice. Standalone CP languages typically ensures KoC through user-provided *select* directives to indicate propagation of choice and a *merging* phase in endpoint projection to verify their consistency. This approach offers flexibility for expressing sophisticated choice propagation patterns, but at the cost of rendering a choreography *unprojectable* if *select* directives are used inconsistently. Library-level CP takes a different approach, where KoC is handled by the library and all choreographies are guaranteed to be projectable. HasChor achieves this by broadcasting the selected choice to all locations, regardless of their participation in the branches. To address the inefficiency of HasChor's approach, *conclaves* [Bates et al. 2024] have been proposed as a way to constrain the scope of conditionals, limiting broadcasts to only the relevant locations.

9 Conclusion

In this paper, we define and study the three variants of freer arrows: freer pre-arrows, freer arrows, and freer choice arrows. These freer arrows are amenable to static analysis. To show why static analysis is useful in Haskell, we conducted a case study on a CP library HasChor. We show that freer arrows help implement static endpoint projection, endpoint static analysis, and selective broadcasting, features that were not possible in the original HasChor.

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. doi:10.1017/S0960129510000496

Mako Bates, Shun Kashiwa, Syed Jafri, Gan Shen, Lindsey Kuper, and Joseph P. Near. 2024. Efficient, Portable, Census-Polymorphic Choreographic Programming. *CoRR* abs/2412.02107 (2024). doi:10.48550/ARXIV.2412.02107 arXiv:2412.02107

Venanzio Capretta. 2005. General recursion via coinductive types. *Log. Methods Comput. Sci.* 1, 2 (2005). doi: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. doi:10.4204/EPTCS.153.2

Marco Carbone and Fabrizio Montesi. 2013. Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 263–274. doi:10.1145/2429069.2429101

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. doi:10.1145/3674625

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2011. On Global Types and Multi-party Sessions. In *Formal Techniques for Distributed Systems*, Roberto Bruni and Juergen Dingel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–28. doi:10.1007/978-3-642-21461-5_1

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. doi: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. doi: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. doi:10.1145/3354166.3354172

Luís Cruz-Filipe and Fabrizio Montesi. 2020. A core model for choreographic programming. *Theoretical Computer Science* 802 (2020), 38–66. doi:10.1016/j.tcs.2019.07.005

Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2013. Meta-theory à la carte. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 207–218. doi:10.1145/2429069.2429094

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. doi:10.22152/programming-journal.org/2019/3/8

Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2024. Choral: Object-oriented Choreographic Programming. *ACM Trans. Program. Lang. Syst.* 46, 1, Article 1 (Jan. 2024), 59 pages. doi:10.1145/3632398

Jeremy Gibbons, Donnacha Oisín Kidney, Tom Schrijvers, and Nicolas Wu. 2023. Phases in Software Architecture. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture* (Seattle, WA, USA) *(FUNARCH 2023)*. Association for Computing Machinery, New York, NY, USA, 29–33. doi:10.1145/3609025.3609479

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. doi: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. doi:10.1007/978-3-031-43345-0_5

Eva Graversen, Andrew K. Hirsch, and Fabrizio Montesi. 2024. Alice or Bob?: Process polymorphism in choreographies. *Journal of Functional Programming* 34 (2024), e1. doi:10.1017/S0956796823000114

Andrew K. Hirsch and Deepak Garg. 2022. Pirouette: Higher-Order Typed Functional Choreographies. *Proc. ACM Program. Lang.* 6, POPL, Article 23 (Jan. 2022), 27 pages. doi:10.1145/3498684

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 273–284. doi:10.1145/1328438.1328472

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. doi:10.1016/S0167-6423(99)00023-4

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. doi: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. doi:10.1145/2804302.2804319

Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C Pierce, and Steve Zdancewic. 2019. From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server. In *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, January 14-15, 2019, Cascais, Portugal*. ACM.

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. doi:10.1145/3704860

Anastasiya Kravchuk-Kirilyuk, Gary Feng, Jonas Iskander, Yizhou Zhang, and Nada Amin. 2024. Persimmon: Nested Family Polymorphism with Extensible Variant Types. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article

119 (April 2024), 27 pages. doi:10.1145/3649836

Robert Krook and Samuel Hammersberg. 2024. Welcome to the Parti(tioning) (Functional Pearl): Using Rewrite Rules and Specialisation to Partition Haskell Programs. 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, 27–40. doi:10.1145/3677999.3678276

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. doi: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. doi:10.1145/3460319.3464798

Yao Li and Stephanie Weirich. 2022. Program adverbs and Tlön embeddings. *Proc. ACM Program. Lang.* 6, ICFP (2022), 312–342. doi: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. doi:10.1016/J.ENTCS.2011.02.018

Lovro Lugović and Sung-Shik Jongmans. 2024. Klor: Choreographies in Clojure. https://github.com/lovrosdu/klor

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. doi:10.1145/2628136.2628144

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. doi: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. doi:10.1017/S0956796807006326

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. doi: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. doi: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. doi:10.1017/S0956796820000088

Fabrizio Montesi. 2013. *Choreographic Programming*. Ph.D. Thesis. IT University of Copenhagen. https://www.fabriziomontesi.com/files/choreographic-programming.pdf.

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. doi: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. doi: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. doi:10.1016/j.entcs.2014.10.015

Dan Plyukhin, Marco Peressotti, and Fabrizio Montesi. 2024. Ozone: Fully Out-of-Order Choreographies. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 31:1–31:28. doi:10.4230/LIPIcs.ECOOP.2024.31

Nick Rioux, Xuejing Huang, Bruno C. d. S. Oliveira, and Steve Zdancewic. 2023. A Bowtie for a Beast: Overloading, Eta Expansion, and Extensible Data Types in F⋈. *Proc. ACM Program. Lang.* 7, POPL (2023), 515–543. doi:10.1145/3571211

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

Takahiro Sanada. 2024. Algebraic effects and handlers for arrows. *J. Funct. Prog.* 34 (2024), e8. doi: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. doi: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. doi: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. doi: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. doi:10.1145/3409003

Wouter Swierstra. 2008. Data types à la carte. *J. Funct. Program.* 18, 4 (2008), 423–436. doi:10.1017/S0956796808006758

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

Philip Wadler. 1998. The Expression Problem. http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt Email correspondence.

Ashton Wiersdorf and Ben Greenman. 2024. Chorex: Choreographic Programming in Elixir. https://github.com/utahplt/chorex

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

The World Wide Web Consortium. 2004. WS Choreography Model Overview. https://www.w3.org/TR/ws-chor-model/

The World Wide Web Consortium. 2005. Web Services Choreography Description Language Version 1.0. https://www.w3.org/TR/ws-cdl-10/

The World Wide Web Consortium. 2006. Web Services Choreography Description Language: Primer. https://www.w3.org/TR/ws-cdl-10-primer/

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. doi: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. 2021a. 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. doi:10.4230/LIPIcs.ITP.2021.32

Weixin Zhang, Yaozhu Sun, and Bruno C. d. S. Oliveira. 2021b. Compositional Programming. *ACM Trans. Program. Lang. Syst.* 43, 3 (2021), 9:1–9:61. doi:10.1145/3460228

A Obtaining Freer 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'] is a method of strong profunctors.
  first (Comp x y) = Comp (first' x) (first' y)
```

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 leads to our definition of *freer pre-arrows*. The key idea is inlining free profunctors in free arrows of Rivas and Jaskelioff, similar to how Kiselyov and Ishii [2015] derive freer monads. This 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.

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`. The definition of `FreerChoiceArrow` can be obtained by inlining both free strong profunctors and free *choice* profunctors.

B A Key-Value Server Choreography Using Freer Arrows

The following is the arrow version of the HasChor key-value store example with a client, primary server, and backup server:

```
discard :: Arrow ar => ar b ()
discard = arr (const ())

asPut :: Request -> Either Request ()
asPut (Put k v) = Left $ Put k v
asPut _ = Right ()

kvs :: Choreo (Kleisli IO) () (Response @ Client)
kvs =
  -- client prepare and send the request
  client `locally0` getRequest >>>
  client ~> primary >>>
  (&&&)
    -- primary handle the request
    (primary `locally` handleRequest)
    -- disparate behavior based on the request
    (cond' primary (arr asPut) $
      (|||)
        -- propogate Put k v
        (arr wrap >>>
        primary ~> backup >>>
        backup `locally` handleRequest >>>
        discard)
        -- do nothing with Get k
        discard
    ) >>>
  arr fst >>>
  primary ~> client
```

The operator `(&&&)` is used to establish that two things happen in parallel: the primary server handles the request, and the primary server forwards the request to the backup server to handle if the request is some `Put k v`. The combinator `cond'` branches on the result of a local computation:

```
cond' :: (Show x, Read x, KnownSymbol l)
      => Proxy l
      -> ar b x -> Choreo ar x a
      -> Choreo ar (b @ l) a
```

Here, the result of the local computation `arr asPut` is `Left` of the original request if it's a `Put` request and `Right` otherwise. This choice is broadcast to the participants of the continuation choreography so that we can branch using `(|||)`. Each branch of `(|||)` receives its data unwrapped, *i.e.,* it has no annotation `@ l`, since it was already broadcast to each endpoint. This means that we have to `wrap` the data to use it to be able to run a local computation on it or send it.

The result of `cond'` here is `()` which we can discard by taking only the result of the left branch of `(&&&)` using `arr fst` before sending the result back to the client.

C Extensible Effects With Freer Arrows

The achieve extensible effects, it's useful to be able to construct compound effects which compose multiple effects together. This approach is used for extensible effects with freer monads Kiselyov and Ishii [2015]. The `Sum2` datatype gives a simple way to do just that; it lets use compose two types of effects as a sum with constructors `Inl2` and `Inr2` resembling the `Left` and `Right` constructors of `Either`. The second class `Inj2` gives us a more flexible way of representing the inclusion of one effect in another by providing a function for injecting an event of one type into the other. With `Inj2`, we can write code that is polymorphic over any effect signature that contains the desired effect.

```
-- |- 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 Datatype for Elgot Iteration

We can reify the Elgot iteration [Adámek et al. 2011; Goncharov 2022] using the following datatype:

```
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
```

The `Elgot` datatype is parameterized by four parameters, an `f`, an `e` of kind Type -> Type -> Type, an `x`, and a `y`. It contains only one constructor, also named `Elgot`. The constructor takes two arguments: a "loop body" of type `f e x (Either z x)` and a continuation of type `f e z y`. The loop body takes an input type `x` and returns a sum type `Either z x`, with a left value of type `z` indicating that the loop has ended, and a right value of type `x`, which is the same type as the input type `x`, indicating that the loop should continue.

We can interpret `Elgot` by combining it with a freer choice arrow:

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

The following function demonstrates that we can define a countdown function using `Elgot` and `FreerChoiceArrow`:

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