Selective Applicative Functors

Declare Your Effects Statically, Select Which to Execute Dynamically

ANDREY MOKHOV, Newcastle University, United Kingdom
GEORGY LUKYANOV, Newcastle University, United Kingdom
SIMON MARLOW, Facebook, United Kingdom
JEREMIE DIMINO, Jane Street, United Kingdom

Applicative functors and monads have conquered the world of functional programming by providing general and powerful ways of describing effectful computations using pure functions. Applicative functors provide a way to compose *independent effects* that cannot depend on values produced by earlier computations, and all of which are declared statically. Monads extend the applicative interface by making it possible to compose *dependent effects*, where the value computed by one effect determines all subsequent effects, dynamically.

This paper introduces an intermediate abstraction called *selective applicative functors* that requires all effects to be declared statically, but provides a way to select which of the effects to execute dynamically. We demonstrate applications of the new abstraction on several examples, including two real-life case studies.

1 INTRODUCTION

Monads, introduced to functional programming by Wadler [1995], are a powerful and general approach for describing effectful (or impure) computations using pure functions. The key ingredient of the monad abstraction is the *bind* operator, denoted by >>= in Haskell[1]:

```
(>>=) :: Monad f => f a -> (a -> f b) -> f b
```

The operator takes two arguments: an effectful computation `f a`, which yields a value of type `a` when executed, and a recipe, i.e. a pure function of type `a -> f b`, for turning `a` into a subsequent computation of type `f b`. This approach to composing effectful computations is inherently sequential: until we execute the effects in `f a`, there is no way of obtaining the computation `f b`, i.e. these computations can only be performed in sequence.

Consider a simple example, where we use the monad `f = IO` to describe an effectful program that prints `"pong"` if the user enters `"ping"`:

```
pingPongM :: IO ()
pingPongM = getLine >>= \s -> if s == "ping" then putStrLn "pong" else pure ()
```

The first argument of the bind operator reads a string using `getLine :: IO String`, and the second argument is the function of type `String -> IO ()`, which prints `"pong"` when `s == "ping"`.

As we will see in sections §3 and §4, in some applications it is desirable to know all possible effects *statically*, i.e. *before the execution*. Alas, this is not possible with monadic effect composition. To *inspect* the function `\s -> ...`, we need an `s`, which becomes available only *during execution*. We are therefore unable to predict the effects that `pingPongM` might perform: instead of conditionally executing `putStrLn`, as intended, it might delete a file from disk, or launch proverbial missiles.

[1]We use Haskell throughout this paper, but the presented ideas are not language specific. We release two libraries for selective applicative functors along with this paper, for Haskell and OCaml, and are also aware of a PureScript fork.

Authors' addresses: Andrey Mokhov, Newcastle University, United Kingdom, andrey.mokhov@ncl.ac.uk; Georgy Lukyanov, Newcastle University, United Kingdom, g.lukyanov2@ncl.ac.uk; Simon Marlow, Facebook, London, United Kingdom, smarlow@fb.com; Jeremie Dimino, Jane Street, London, United Kingdom, jdimino@janestreet.com.

2019. 2475-1421/2019/1-ART1 $15.00
https://doi.org/

Applicative functors, introduced by McBride and Paterson [2008], can be used for composing statically known collections of effectful computations, as long as these computations are *independent* from each other. The key ingredient of applicative functors is the *apply* operator, denoted by `<*>`:

```
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

The operator takes two effectful computations, which — independently — compute values of types `a -> b` and `a`, and returns their composition that performs both computations, and then applies the obtained function to the obtained value producing the result of type `b`. Crucially, both arguments and associated effects are known statically, which, for example, allows us to pre-allocate all necessary computation resources upfront (§3) and execute all computations in parallel (§4).

Our ping-pong example cannot be expressed using applicative functors. Since the two computations must be independent, the best we can do is to print `"pong"` unconditionally:

```
pingPongA = fmap (\s -> id) getLine <*> putStrLn "pong"
```

We use `fmap (\s -> id)` to replace the input string `s`, which we now have no need for, with the identity function `id :: () -> ()`, thus matching the return type of `putStrLn "pong" :: IO ()`. We cannot execute the `putStrLn` effect conditionally but, on the positive side, the effects are no longer hidden behind opaque effect-generating functions, which makes it possible for the applicative functor `f = IO` to statically know the two effects embedded in `pingPongA`.

At this point the reader is hopefully wondering: can we combine the advantages of applicative functors and monads, i.e. allow for conditional execution of some effects while retaining the ability to statically know all effects embedded in a computation? It will hardly be a surprise that the answer is positive, but it is far from obvious what the right abstraction should be. For example, one might consider adding a new primitive called `whenS` to `IO`:

```
whenS :: IO Bool -> IO () -> IO ()
```

This primitive executes the first computation, and then uses the obtained `Bool` to decide whether to execute the second computation or not. Let us rewrite the ping-pong example using `whenS`:

```
pingPongS = whenS (fmap (=="ping") getLine) (putStrLn "pong")
```

We replace the input string `s` with `True` if it is equal to `"ping"`, and `False` otherwise, thereby appropriately *selecting* the subsequent effectful computation. This approach gives us both conditional execution of `putStrLn "pong"`, and static visibility of both effects (see §5.2). Crucially, `whenS` must be an `IO` primitive instead of being implemented in terms of the monadic bind (`>>=`), because the latter would result in wrapping `putStrLn` into an opaque function, as in `pingPongM`.

The main idea of this paper is that `whenS`, as well as many other similar combinators, can be seen as special cases of a new intermediate abstraction, called *selective applicative functors*, whose main operator for composing effectful computations is *select*:

```
select :: Selective f => f (Either a b) -> f (a -> b) -> f b
```

Intuitively, the first effectful computation is used to select what happens next: if it yields a `Left a` you *must execute* the second computation in order to produce a `b` in the end; otherwise, if it yields a `Right b`, you *may skip* the subsequent effect, because you have no use for the resulting function.

The contributions of this paper are as follows:

- We introduce *selective applicative functors* as a general abstraction situated between applicative functors and monads, characterising the relationships between all three abstractions with a set of laws, and defining a few important instances (§2).
- We discuss applications of the abstraction on two real-life case studies: the OCaml build system Dune [Jane Street 2018] (§3) and Facebook's Haxl library [Marlow et al. 2014] (§4).
- We present *free selective applicative functors* and show how to use them to implement embedded domain-specific languages with both conditional effects and static analysis (§5).

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

-- An infix synonym for fmap
(<$>) :: Functor f => (a -> b) -> f a -> f b       -- (<$>) is pronounced "fmap"
```

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b              -- (<*>) is pronounced "apply"

-- A variant of (<*>) that discards the value of the first argument
(*>) :: Applicative f => f a -> f b -> f b
```

```
class Applicative f => Selective f where
    select :: f (Either a b) -> f (a -> b) -> f b

-- An infix synonym for select
(<*?) :: f (Either a b) -> f (a -> b) -> f b       -- (<*?) is pronounced "select"
```

```
class Selective f => Monad f where
    return :: a -> f a                             -- Note the law: return = pure
    (>>=)  :: f a -> (a -> f b) -> f b             -- (>>=) is pronounced "bind"

-- A monadic equivalent of the apply operator, satisfying the law (<*>) = ap
ap :: Monad f => f (a -> b) -> f a -> f b
```

Fig. 1. The proposed type class hierarchy, where `Functor`, `Applicative` and `Monad` are standard Haskell type classes, and `Selective` is a new intermediate abstraction introduced between `Applicative` and `Monad`.

We discuss alternatives to selective applicative functors and related work in sections §6 and §7.

2 SELECTIVE FUNCTORS

In this section we introduce selective applicative functors, which we will subsequently refer to as simply *selective functors*, for brevity. We start by defining the new abstraction, and then use it in §2.1 to implement several derived combinators, such as the aforementioned `whenS`. In §2.2 we provide several examples of selective functors, and discuss the relationships between applicative functors, selective functors, and monads. In §2.3, these relationships are further elaborated and expressed as a set of laws that all selective functors are required to satisfy.

Like applicative functors [McBride and Paterson 2008], selective functors provide a way to embed pure values into an effectful context `f` using the function `pure`, and give meaning to composition of two *independent* effectful computations using the operator `<*>`. See Fig. 1 for the standard definition of the corresponding type class `Applicative`. Selective functors enrich the applicative interface with the `select` function, which gives meaning to the composition of two effectful computations, where, in contrast to `<*>`, the second computation *depends* on the first one:

```
class Applicative f => Selective f where
    select :: f (Either a b) -> f (a -> b) -> f b
```

One can think of `select` as a selective function application: parametricity [Wadler 1989] dictates that, when given a `Left a`, we *must execute the effects* in `f (a -> b)`, apply the obtained function to `a`, and return the resulting `b`; on the other hand, when given a `Right b`, we *may skip the effects* associated with the function, and return the given `b`.

```
($)     :: (a -> b) -> a -> b                                -- Function application
(.)     :: (b -> c) -> (a -> b) -> a -> c                    -- Function composition
id      :: a -> a                                           -- Identity function
const   :: a -> b -> a                                      -- Constant function
flip    :: (a -> b -> c) -> b -> a -> c                     -- Flip function arguments
uncurry :: (a -> b -> c) -> (a, b) -> c                     -- Uncurry a function
foldr   :: (a -> b -> b) -> b -> [a] -> b                   -- Reduce a list to a value
bool    :: a -> a -> Bool -> a                              -- Deconstruct a Bool
maybe   :: b -> (a -> b) -> Maybe a -> b                    -- Deconstruct a Maybe
either  :: (a -> c) -> (b -> c) -> Either a b -> c          -- Deconstruct an Either
first   :: (a -> c) -> Either a b -> Either c b             -- Map over Left
second  :: (b -> c) -> Either a b -> Either a c             -- Map over Right
bimap   :: (a -> c) -> (b -> d) -> Either a b -> Either c d -- Map over Left and Right
void    :: Functor f => f a -> f ()                         -- Discard an effect's value
```

Fig. 2. Type signatures and descriptions of standard operators and functions used throughout the paper.

Following the notational convention for applicative operators, we also define the infix operator alias `<*?` for `select`: the angle bracket pointing to the left means we always use the corresponding value; the value on the right, however, may be skipped, hence the question mark.

One can implement `select` using monads in a straightforward manner: examine the value produced by `f` (`Either a b`) with the bind operator, and then, in the `Left a` case, execute the subsequent effect `f (a -> b)`, passing the `a` to it using the `Functor`'s map operator, as shown below. We will use `<$>`, an infix synonym of `fmap`, throughout the paper (see Fig. 1 for `Functor`'s API).

```
selectM :: Monad f => f (Either a b) -> f (a -> b) -> f b
selectM x y = x >>= \e -> case e of Left  a -> ($a) <$> y -- Execute y
                                    Right b -> pure b     -- Skip y
```

Many monads directly use `select = selectM` in their `Selective` instance definitions, and in §2.3 we argue that this should in fact be a law when both `Selective f` and `Monad f` instances exist. Note that some monads, e.g. the Haxl monad (§4), choose to implement the `select` method differently for performance reasons, but they still satisfy the law `select = selectM` at the semantic level.

One can also implement a function with the type signature of `select` using applicative functors, but it will always execute the effects associated with the second argument, rendering any conditional execution of effects impossible, as in the `pingPongA` example in §1:

```
selectA :: Applicative f => f (Either a b) -> f (a -> b) -> f b
selectA x y = (\e f -> either f id e) <$> x <*> y -- Execute x and y
```

Fig. 2 gives type signatures and short descriptions for standard functions `either`, `id`, and other convenient functional combinators that we use in this paper.

While `selectM` is useful for *conditional execution* of effects, `selectA` is useful for *static analysis*. As we will see in §2.2, selective functors used for static analysis need to collect information about all possible effects instead of skipping some of them, hence they directly use `select = selectA` in their `Selective` instance definitions.

Any `Applicative` instance can thus be given a `Selective` instance. The opposite is also true in the sense that one can recover the operator `<*>` from `select` as follows:

```
apS :: Selective f => f (a -> b) -> f a -> f b
apS f x = select (Left <$> f) (flip ($) <$> x)
```

Here we tag a given function `a -> b` with `Left` and turn a value of type `a` into the reverse application function `\a f -> f a`, which yields `b` when given `a -> b`, as desired. Since the `Right` case is impossible, the effect `f a` is executed unconditionally. Note however, that the equality `(<*>) = apS` does not always hold. Selective functors that satisfy the law `(<*>) = apS` will be called *rigid*; they will turn out to have a particularly simple normal form, which we will exploit in the free construction in §5.

We will come back to the relationship between applicative functors, selective functors and monads in §2.3, after first exploring *selective combinators* that can be written using the selective interface (§2.1), and looking at some concrete examples of selective functors (§2.2).

2.1 Selective combinators

As a first use-case of the interface provided by selective functors, let us revisit our ping-pong example from §1 and implement the combinator `whenS`:

```
whenS :: Selective f => f Bool -> f () -> f ()
whenS x y = selector <*? effect
  where
    selector = bool (Right ()) (Left ()) <$> x -- NB: convert True to Left ()
    effect   = const                     <$> y
```

We first bring the given effectful computations into the right shape by using the `Functor`'s map operator. Specifically, `x :: f Bool` is converted into a `selector :: f (Either () ())`, and `y :: f ()` is converted into `effect :: f (() -> ())`. The results are composed using the select operator `<*?`, and the meaning of this composition is determined by the supplied `Selective f` instance. For example, an instance like `f = IO` would skip `y` if `x` yields `False`, as exploited by our implementation of `pingPongS`. On the other hand, instances used for static analysis would record both `x` and `y` as possible effects. See more examples in §2.2.

It is worth noting that unlike the select operator, whose implementation is almost completely determined by parametricity (i.e., the only real question is: *"To skip, or not to skip?"*), `whenS` admits a variety of (incorrect) implementations. In particular, due to *Boolean blindness*[2], it is easy to inadvertently implement `unlessS`, which has the same type but flips the meaning of the Boolean value. The ability to reason parametrically was one of the guiding principles we used when looking for a good abstraction for selective functors: `select` provides this ability, whereas `whenS` does not.

A strong contender for playing the leading role in selective functors is the function `branch` that, given an effectful computation `x :: f (Either a b)`, selects which subsequent computation, namely `l :: f (a -> c)` or `r :: f (b -> c)`, to execute:

```
branch :: Selective f => f (Either a b) -> f (a -> c) -> f (b -> c) -> f c
branch x l r = fmap (fmap Left) x <*? fmap (fmap Right) l <*? r
```

While we encourage the reader to derive an implementation of `branch` as an exercise, we would like to share our intuition behind it, as it will be useful for *free selective functors* in §5. The select operator allows us to eliminate one of the cases in a sum type, namely the `Left a` case in `Either a b`, leaving the other case intact. To implement `branch`, we will need to apply `<*?` twice, eliminating `a` and `b` one after another. The first application is tricky because `f (Either a b)` and `f (a -> c)` do not match the type signature of `<*?`. To fix the mismatch, we convert them to `f (Either a (Either b c))` and `f (a -> Either b c)`, respectively. The second application of `<*?` is then straightforward.

[2]The term refers to the fact that the `True` and `False` values are not distinguished at the type level, see Diehl [2018].

```
whenS :: Selective f => f Bool -> f () -> f ()
whenS x y = select (bool (Right ()) (Left ()) <$> x) (const <$> y)

branch :: Selective f => f (Either a b) -> f (a -> c) -> f (b -> c) -> f c
branch x l r = fmap (fmap Left) x <*? fmap (fmap Right) l <*? r

ifS :: Selective f => f Bool -> f a -> f a -> f a
ifS x t e = branch (bool (Right ()) (Left ()) <$> x) (const <$> t) (const <$> e)

(<||>) :: Selective f => f Bool -> f Bool -> f Bool
x <||> y = ifS x (pure True) y

(<&&>) :: Selective f => f Bool -> f Bool -> f Bool
x <&&> y = ifS x y (pure False)

fromMaybeS :: Selective f => f a -> f (Maybe a) -> f a
fromMaybeS x mx = select (maybe (Left ()) Right <$> mx) (const <$> x)

anyS :: Selective f => (a -> f Bool) -> [a] -> f Bool
anyS p = foldr ((<||>) . p) (pure False)

allS :: Selective f => (a -> f Bool) -> [a] -> f Bool
allS p = foldr ((<&&>) . p) (pure True)

whileS :: Selective f => f Bool -> f () -- Run a computation while it yields True
whileS x = whenS x (whileS x)
```

Fig. 3. A library of selective combinators. The names and order of parameters are inherited from the standard Haskell library. For example, `fromMaybeS` corresponds to the standard `fromMaybe :: a -> Maybe a -> a` and retains the short-circuiting behaviour, i.e. if the second argument yields a `Just`, the first argument is skipped.

As will be discussed in §6.1, we could have chosen to use `branch` instead of `select` as the method of the `Selective` type class. Our choice of `select` follows the Occam's razor principle: `select` is simpler than `branch`, which, in particular, leads to a simpler free construction (§5.1).

By instantiating `select` with `a = b = ()` we have earlier obtained `whenS`. Below we repeat the same trick but with `branch`, obtaining another familiar conditional combinator `ifS`:

```
ifS :: Selective f => f Bool -> f a -> f a -> f a
ifS x t e = branch selector (const <$> t) (const <$> e)
  where
    selector = bool (Right ()) (Left ()) <$> x -- NB: convert True to Left ()
```

Many conditional combinators, which are typically associated with the `Monad` type class, can be expressed using selective functors, as shown in Fig. 3, making them reusable in new contexts. In particular, the logical combinators `<||>` and `<&&>` will play an important role in improving the efficiency of the Haxl framework in §4. To emphasise the monadic flavour of selective functors, we can rewrite `ifS` in the form of the bind operator specialised to `Bool`:

```
bindBool :: Selective f => f Bool -> (Bool -> f a) -> f a
bindBool x f = ifS x (f False) (f True)
```

This can be achieved not only for `Bool`, but for any *enumerable* type, as we will discuss in §6.1.

2.2 Examples of selective functors

Having explored various useful combinators that can be implemented on top of the minimalistic selective interface, in this section we look at several examples of selective functors.

As we have observed at the beginning of this section, any monad can be given a `Selective` instance simply by using `select = selectM` as the definition. As an example, below we define a selective instance for `IO`, and test the function `pingPongS` from §1 in an interactive GHC session:

```
instance Selective IO where
    select = selectM

λ> pingPongS = whenS ((=="ping") <$> getLine) (putStrLn "pong")
λ> pingPongS
hello
λ> pingPongS
ping
pong
```

As desired, the effect `putStrLn "pong"` occurs conditionally. In §5.2 we will see how to statically analyse `pingPongS`, i.e. obtain a list of all possible effects *before the execution*, which can be done by interpreting `pingPongS` in the selective functor `Over`, introduced below.

The `Const m a` functor is an interesting instance of the `Applicative` type class, which stores no values of type `a`, but keeps track of the applicative structure in the monoid value of type `m`:

```
newtype Const m a = Const { getConst :: m }

instance Functor (Const m) where
    fmap _ (Const x) = Const x

-- 'mempty' and '<>' are the identity and the binary operation of the Monoid m
instance Monoid m => Applicative (Const m) where
    pure _              = Const mempty   -- Pure values have no effects
    Const x <*> Const y = Const (x <> y) -- Collect effects in x and y
```

It turns out there are two useful selective instances for `Const`. To disambiguate between them, we will call them `Over` and `Under`, reusing[3] the above `Functor` and `Applicative` instances:

```
newtype Over  m a = Over  { getOver  :: m }
newtype Under m a = Under { getUnder :: m }

instance Monoid m => Selective (Over m) where
    select (Over x) (Over y) = Over (x <> y) -- Collect effects in x and y

instance Monoid m => Selective (Under m) where
    select (Under x) _ = Under x             -- Discard conditional effects
```

The selective functor `Over` can be used for computing a list of all possible effects embedded in a computation, i.e. an *over-approximation* of the effects that will actually occur. This is achieved by keeping track of effects in both arguments of `select`. The selective functor `Under`, on the other hand, discards the second argument of `select`, and therefore computes an *under-approximation*, i.e. a list of effects that are guaranteed to occur.

[3] Fortunately, thanks to the new GHC extension DerivingVia [Blöndal et al. 2018], we can reuse `Const` instances without duplicating any code, simply by adding `deriving (Functor, Applicative) via (Const m)` to the `newtype` definitions.

Let us give these two instances a try:

```
λ> ifS (Over "a") (Over "b") (Over "c") *> Over "d" *> whenS (Over "e") (Over "f")
Over "abcdef"

λ> ifS (Under "a") (Under "b") (Under "c") *> Under "d" *> whenS (Under "e") (Under "f")
Under "ade"
```

As expected, **Over** collects all effects, whereas **Under** does not look beyond "opaque" conditions. A deeper difference between them is that **Over** is a rigid selective functor, i.e. `(<*>) = apS`, but **Under** is not: indeed, `Under "a" <*> Under "b"` records both `"a"` and `"b"`, but `apS (Under "a") (Under "b")` records just `"a"` since `apS` is implemented via `select`. Intuitively, non-rigid selective functors have a much richer structure, because `<*>` is not expressible via `select`, which significantly complicates the corresponding free construction (§5.1).

Our last example in this section is the selective functor **Validation**[4], which is useful for validating complex data: if reading one or more data fields has failed, all errors are accumulated (using the operator `<>` from the semigroup `e`) to be reported together.

```
data Validation e a = Failure e | Success a deriving Functor

instance Semigroup e => Applicative (Validation e) where
    pure = Success
    Failure e1 <*> Failure e2 = Failure (e1 <> e2) -- Accumulate errors
    Failure e1 <*> Success _  = Failure e1
    Success _  <*> Failure e2 = Failure e2
    Success f  <*> Success a  = Success (f a)
```

The error-accumulating behaviour cannot be extended to a **Monad** instance, since the bind operator has nothing to bind to if the first computation fails. We can, however, define the following **Selective** instance, which allows us to validate data in the presence of conditions.

```
instance Semigroup e => Selective (Validation e) where
    select (Success (Right b)) _ = Success b
    select (Success (Left  a)) f = ($a) <$> f
    select (Failure e) _ = Failure e -- Discard errors behind failed conditions
```

Here, the last line is particularly interesting: similarly to **Under**, we discard the second argument, but *only if the first has failed*. This allows us not to report any validation errors that are hidden behind a failed conditional (we do not know whether the corresponding fields are actually needed).

Below we define a function that constructs a shape (a circle or a rectangle) given a choice of the shape `x`, and the shape's parameters (`r`, `w`, and `h`) in an arbitrary selective functor `f`. You can think of the inputs as results of reading the corresponding fields from a web form, where `x` is a checkbox, and all other fields are numeric textboxes, some of which may be empty.

```
type Radius = Word; type Width = Word; type Height = Word

data Shape = Circle Radius | Rectangle Width Height

shape :: Selective f => f Bool -> f Radius -> f Width -> f Height -> f Shape
shape x r w h = ifS x (Circle <$> r) (Rectangle <$> w <*> h)
```

We choose `f = Validation [String]` to report the errors that occurred when reading values from the form. Let us see how this works.

[4] Applicative functors **Const** and **Validation** appeared under the names **Accy** and **Except** in [McBride and Paterson 2008].

```
λ> shape (Success True) (Success 1) (Failure ["width?"]) (Failure ["height?"])
Success (Circle 1)
λ> shape (Success False) (Failure ["radius?"]) (Success 2) (Success 3)
Success (Rectangle 2 3)
λ> shape (Success False) (Success 1) (Failure ["width?"]) (Failure ["height?"])
Failure ["width?", "height?"]
λ> shape (Failure ["choice?"]) (Failure ["radius?"]) (Success 2) (Failure ["height?"])
Failure ["choice?"]
```

In the last example, since the shape's choice could not be read, we do not report any subsequent errors. But it does not mean we are short-circuiting the validation: we will continue accumulating errors as soon as we get out of the failed conditional, as demonstrated below.

```
twoShapes :: Selective f => f Shape -> f Shape -> f (Shape, Shape)
twoShapes s1 s2 = (,) <$> s1 <*> s2

λ> s1 = shape (Failure ["choice 1?"]) (Success 1) (Failure ["width 1?"]) (Success 3)
λ> s2 = shape (Success False) (Success 1) (Success 2) (Failure ["height 2?"])
λ> twoShapes s1 s2
Failure ["choice 1?","height 2?"]
```

Like `Under`, the `Validation` instance is not a rigid selective functor, since `select` occasionally discards effects in the second argument.

2.3 Laws

Now that we have seen several instances of selective functors, it is time to discuss the laws that we expect these instances to satisfy. In particular, it is very useful to know how the selective interface interacts with existing applicative and monadic interfaces. Real code might mix all of these abstractions, because each of them is useful in its own right, and the laws presented in this section allow us to safely refactor such code while keeping its original meaning.

Fig. 4 lists all laws for selective functors, as well as some useful theorems that can be derived from the laws and parametricity [Wadler 1989]. Below we discuss them in order.

The *identity* and *distributivity* laws determine the interaction of the select operator with pure computations. It should be impossible to distinguish `x <*? pure id` from a direct execution of `x` followed by the extraction of a value from the obtained `Either a a`. In particular, the select operator is not allowed to duplicate effects associated with `x`. Similarly, the select operator is not allowed to sneak in any effects if the first computation is pure, which allows `pure x <*?` to be distributed through the applicative sequencing operator `<*>`. When applied in reverse, the distributivity law allows us to simplify sequences of conditional operations *as long as the conditions are pure*. The *generalised identity* theorem follows from the identity law by parametricity.

Note that it is not a requirement for selective functors to skip unnecessary effects. In particular, we do not require that `pure (Right x) <*? y = pure x`. It may be counterintuitive, but omitting this law makes selective functors more useful. Typically, *when executing* a selective computation, you would want to skip the unnecessary effects (saving work); but on the other hand, if your goal is to *statically analyse* a given selective computation and extract the set of all possible effects (without actually executing them), then you do not want to skip any effects, because this would defeat the purpose of static analysis. `Over` is an example of a selective functor that would violate the requirement to skip unnecessary effects. Similarly, we do not require that `pure (Left x) <*? y = ($x) <$> y` thus legalising under-approximating instances like `Under` and `Validation`. It is worth noting, however, that the monadic select operator `selectM` does satisfy the *pure Left* and *pure Right* properties.

```
-- Identity
x <*? pure id = either id id <$> x

-- Distributivity; note that y and z have the same type f (a -> b)
pure x <*? (y *> z) = (pure x <*? y) *> (pure x <*? z)

-- Associativity
x <*? (y <*? z) = (f <$> x) <*? (g <$> y) <*? (h <$> z)
  where
    f x = Right <$> x
    g y = \a -> bimap (,a) ($a) y
    h z = uncurry z

-- For selective functors that are also monads:
select = selectM
```

```
-- Apply a pure function to the result
f <$> select x y = select (fmap f <$> x) (fmap f <$> y)

-- Apply a pure function to the Left case of the first argument
select (first f <$> x) y = select x ((. f) <$> y)

-- Apply a pure function to the second argument
select x (f <$> y) = select (first (flip f) <$> x) (flip ($) <$> y)

-- Generalised identity
x <*? pure y = either y id <$> x

-- Laws for rigid selective functors (in particular, for monads)
(<*>) = apS                      -- Selective apply
x *> (y <*? z) = (x *> y) <*? z -- Interchange
```

Fig. 4. Laws (top) and theorems (bottom) of selective functors. Coq proofs that the selective instances from §2.2 are lawful are available in supplementary material.

Such loose requirements on `select` with respect to unnecessary effects might seem troublesome. Below we list three reasons why we chose to keep the requirements loose.

- Requiring unnecessary effects to be skipped rules out any instances that could be used for static analysis. Indeed, the only way to obey this law would be to look at actual values at runtime, since it is impossible to statically know if `f (Either a b)` contains a **`Right`** value.
- One might suggest having two classes **`Selective`** and **`StrictSelective`**, the latter with stricter requirements. But this second class would be useless: to statically analyse a computation you would have to express it via **`Selective`**, since it is inhabited by instances like **`Over`**. As for the execution, a **`Monad`** is perfectly suitable, as we will see in §5.2.
- Finally, there is a good precedent: the **`Applicative`** type class has no requirements on the order in which effects are executed: left-to-right, right-to-left, or in parallel. This loose specification allows some instances to execute effects sequentially (typically from left to right), and other instances to execute effects in parallel. Note that as soon as we also have a **`Monad`**, we gain an additional requirement `(<*>) = ap`, which tells us that, at the semantic level, the result should be as if the effects were executed in sequence from left to right. We follow the same approach by requiring `select = selectM` if `f` is also a **`Monad`** (see below).

The *associativity* law states that it should always be possible to re-associate a sequence of select operators to the left, by doing the necessary adjustments to shapes of the inner values. These adjustments, called `f`, `g`, and `h` in Fig. 4, are admittedly obscure and require an explanation. For the expression `x <*? (y <*? z)` to typecheck, the arguments should have the following types:

```
x :: f (Either a b)          y :: f (Either c (a -> b))     z :: f (c -> a -> b)
```

On the other hand, the resulting expression `p <*? q <*? r` has arguments of these types:

```
p :: f (Either a (Either (c,a) b))   q :: f (a -> Either (c,a) b)   r :: f ((c,a) -> b)
```

To adjust `x`, we inject it in a larger sum type, `y` is turned into a function accepting a value of type `a` from `p`, and `z` is simply `uncurry`ed. As we will see in §6.2, the associativity law can be expressed much more naturally if we switch to a more symmetric select operator (a similar phenomenon occurs with the *composition* law of the **`Applicative`** type class when the latter is expressed using an equivalent but more symmetric **`Monoidal`** interface [McBride and Paterson 2008]).

The final law links selective functors to monads: in the spirit of the conventional applicative-monad law `(<*>) = ap`, we require that monadic instances implement `select` so that it is extensionally equivalent to `selectM`, which in particular means that unnecessary effects are skipped. A consequence of this law is that monadic selective functors are also rigid, i.e. `(<*>) = apS`, which makes it practically feasible to reason about code written using all three abstractions.

Fig. 4 also lists a few theorems that are useful when working with selective functors. The first three come for free from parametricity: they tell how one can reshape pure contents of selective functors. Last but not least, the *interchange* property is a consequence of associativity and `(<*>) = apS`, which allows to move computations inside the condition argument of the select operator. This property does not always hold for non-rigid selective functors: while **`Under`** respects it, **`Validation`** does not, as demonstrated by the following example:

```
λ> whenS (Under "a" *> Under "b") (Under "c")
Under "ab"
λ> Under "a" *> whenS (Under "b") (Under "c")
Under "ab"

λ> Failure "a" *> whenS (Success True) (Failure "b")
Failure "ab"
λ> whenS (Failure "a" *> Success True) (Failure "b")
Failure "a"
```

One example where the interchange law appears naturally is parser combinators, where it allows us to refactor parsers with choice, see §7.2.

3 STATIC ANALYSIS

In this section we discuss a real-life application that benefits from static analysis of effectful computations – the Dune build system [Jane Street 2018]. We start by introducing Dune and motivating the need for static analysis with over-approximation (§3.1), and then show how one can implement static analysis of build system dependencies using selective functors (§3.2).

3.1 Dune build system

Dune was originally developed at Jane Street and has by now become a standard build system for OCaml packages [Jane Street 2018]. At the time of writing, more than 1000 OCaml packages are using Dune as the build system. The original motivation for developing Dune (earlier known as jbuilder) was to make it easier to open source code developed in an industrial environment, and so Dune was not meant to be used for everyday software development. However, Dune's ability to

extract maximum parallelism from build scripts meant it was faster than existing build systems, such as OCamlbuild, and it quickly became popular, with major projects switching to Dune, for example, the Coq proof assistant [Bertot and Castéran 2013].

One unusual feature of Dune is the ability to statically over-approximate all build dependencies of a package. This is used at Jane Street to automatically produce *package manifest files* for more than 100 packages instead of maintaining them by hand. Package manifest files are consumed by *package managers*, such as OPAM [The OPAM team 2018], which download and install all required dependencies *before the build starts*.

To generate a manifest file automatically Dune needs to analyse the build graph statically, i.e. *without actually running any build commands*, because at this point the project cannot yet be built (due to missing dependencies). Package dependencies can be conditional and depend on values that can only be computed during the build, therefore in many situations it is impossible to statically compute an exact set of dependencies, and hence an over-approximation is used instead.

In general, one can view such static dependency analysis as a function from a build script to a set of package dependencies, and implement it directly by parsing the script and extracting all possible dependencies from it. Dune adopts a different approach: it reuses the existing script execution engine that executes build commands, but in a mock environment where commands are skipped, but their dependencies are recorded in all branches of conditional statements. By doing static analysis at this level, one can reuse a lot of code, e.g. for parsing and interpreting build scripts.

In this mock environment, some parts of the code cannot be fully evaluated as they need the output produced by external commands. However, these parts still need to be analysed. To achieve this, the original implementation of Dune uses the *arrow* abstraction discussed in §7.1. To evaluate suitability of selective functors for this task, we have successfully prototyped an alternative core for Dune, which uses applicative and selective functors instead of arrows.

3.2 Static analysis of build dependencies

Dune is written in OCaml, and we therefore developed an OCaml library for selective functors. In this section, however, we choose to continue using Haskell to avoid confusion.

We follow the approach by Mokhov et al. [2018] for modelling *build tasks*, where a single task is represented as a higher-order effectful function parameterised by the type of *keys* `k`, e.g. file names, and the type of *values* `v`, e.g. file contents. A task takes a *callback* of type `k -> f v`, that the task can use to find values of its dependencies, and returns the result embedded in a selective context `f`:

```
newtype Task k v = Task { run :: forall f. Selective f => (k -> f v) -> f v }
```

The task needs to be polymorphic over the `f` so that it can be run both in *build mode*, by actually executing build commands, and in the *mock mode*, where build commands are skipped but dependencies are recorded, as explained in §3.1. For example, to compute over- and under-approximation of build dependencies we can run the task in selective functors `f = Over` and `f = Under`, respectively:

```
dependenciesOver :: Task k v -> [k]
dependenciesOver task = getOver $ run task (\k -> Over [k])

dependenciesUnder :: Task k v -> [k]
dependenciesUnder task = getUnder $ run task (\k -> Under [k])
```

Thanks to the polymorphism of the task description over `f`, we can "execute" a given task with a mock callback like `(\k -> Over [k]) :: k -> Over v`, whose only effect is recording the given key.

To demonstrate this on an example, we need a way to model a *build script*, i.e. a collection of build tasks. One simple approach [Mokhov et al. 2018] is to use a function that, given a key `k` returns

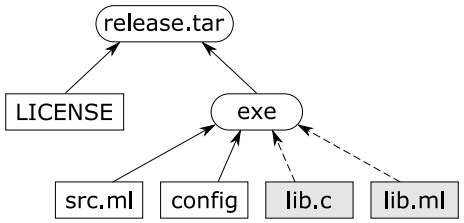


Fig. 5. An example build dependency graph. Input files are shown in rectangles, intermediate and output files are shown in rounded rectangles. Approximate dependencies are highlighted with dashed lines.

either the corresponding build `Task` or `Nothing` to indicate that this key is an input (external) dependency that cannot be built and should therefore be available before the build starts:

```
type Script k v = k -> Maybe (Task k v)
```

Now we have all the ingredients for creating a simple build script comprising two tasks: (i) the top-level task for building release.tar by archiving the file LICENSE and the executable exe; and (ii) the task for compiling the executable from the source src.ml and one of the two libraries: lib.c or lib.ml, depending on the configuration option stored in the config file (it is common to use an optimised low-level C implementation of a performance-critical function, falling back to high-level OCaml implementation if the former is unavailable on the system):

```
script :: Script FilePath String
script "release.tar" = Just $ Task $ \fetch -> tar [fetch "LICENSE", fetch "exe"]
script "exe" = Just $ Task $ \fetch ->
    let src   = fetch "src.ml"
        cfg   = fetch "config"
        libc  = fetch "lib.c"
        libml = fetch "lib.ml"
    in compile [src, ifS (parse cfg) libc libml]
script _ = Nothing
```

We assume the existence of functions `tar` (creating an archive), `compile` (compiling an OCaml executable from sources and libraries), and `parse` (parsing a configuration file); their implementation is irrelevant for our purposes.

By analysing individual build tasks using `dependenciesOver` and `dependenciesUnder`, we can obtain an approximate build dependency information for the whole dependency graph:

```
λ> dependenciesOver (fromJust $ script "release.tar")
["LICENSE","exe"]
λ> dependenciesUnder (fromJust $ script "release.tar")
["LICENSE","exe"]
λ> dependenciesOver (fromJust $ script "exe")
["src.ml","config","lib.c","lib.ml"]
λ> dependenciesUnder (fromJust $ script "exe")
["src.ml","config"]
```

Fig. 5 visualises the graph obtained by traversing approximate dependencies starting from the target key release.tar.

Note that while over-approximation is useful for installing all possible dependencies *before the build*, under-approximation is useful for maximising parallelism *during the build*: for example, if all input files in our example are actually generated, e.g. by running a text preprocessor, then we can start the three preprocessing tasks that are definitely needed (LICENSE, src.ml, config) in parallel, i.e. without waiting for the outcome of parsing the config file.

Applicative and monadic build systems studied in [Mokhov et al. 2018] cannot support such over- and under-approximating static analysis, and the associated abstractions are therefore unsuitable for Dune. This explains why Dune developers have chosen to use the *arrow* abstraction (§7.1). As our case study and the developed prototype demonstrate, selective functors provide a viable alternative to arrows in the context of build systems.

4 SPECULATIVE EXECUTION

Haxl [Marlow et al. 2014] is a framework for efficiently executing code that fetches data from external sources, typically databases or remote services. The Haxl framework allows code written in a natural style using **Applicative** and **Monad** combinators to run efficiently, by automatically parallelising the data fetch operations and batching together multiple fetches from the same data source. Haxl has been in use at Facebook, at scale, for several years now in a system that proactively detects and remediates various forms of abuse. Haxl allows the engineers working on the anti-abuse code to write clear and concise application logic, because the framework abstracts away from the details of concurrency and efficient data-fetching.

To illustrate the idea using a fragment of the example code from [Marlow et al. 2014], suppose we are writing the code to render a blog into HTML. The blog consists of a set of posts, where each post is identified by a **PostId**. The data for the blog is stored in a remote database. The API for fetching the data from the database is as follows:

```
getPostIds     :: Haxl [PostId]
getPostContent :: PostId -> Haxl PostContent
```

We can fetch the set of all **PostId**s using `getPostIds`, and we can fetch the content of one post using `getPostContent`. To get the content of all posts we could write:

```
getAllPostsContent :: Haxl [PostContent]
getAllPostsContent = getPostIds >>= mapM getPostContent
```

Now, when we `mapM getPostContent` we would really like the database queries to happen in parallel, because there are no dependencies between them. Furthermore, we might even be able to batch up the queries into a single request to the remote database.

These optimisations are performed automatically by Haxl, using a special **Applicative** instance that exploits the lack of dependency between the two computations to explore the computations and collect the data fetch operations that can be performed in parallel or batched together. Haxl also has a **Monad** instance that sequentialises operations as you would expect, but code written to use **Applicative** operations benefits from the automatic concurrency. This optimisation is further exploited by using a transformation on the monadic **do** syntax to automatically use **Applicative** operators where possible [Marlow et al. 2016].

One of the key tools found to be useful in the kind of code written using Haxl in production is the "lazy" conditional operators:

```
(.||), (.&&) :: Haxl Bool -> Haxl Bool -> Haxl Bool
x .|| y = do b <- x; if b then return True else y
x .&& y = do b <- x; if b then y           else return False
```

```
instance Selective Haxl where
    select (Haxl x) (Haxl f) = Haxl $ do
        rx <- x
        case rx of
            Done (Right b) -> return (Done b)
            Done (Left  a) -> unHaxl (($a) <$> Haxl f)
            Throw e        -> return (Throw e)
            Blocked bx c   -> do
                rf <- f
                case rf of
                    Done f  -> unHaxl (either f id <$> c)
                    Throw e -> return (Blocked bx (select c (Haxl (return rf))))
                    Blocked by d -> return (Blocked (bx <> by) (select c d)
```

Fig. 6. An implementation of `Selective` instance for Haxl.

These are typically used to improve performance by guarding slow checks with faster checks. For example, we might write:

```
if simpleCondition .&& complexCondition then ... else ...
```

The idea is that `simpleCondition` is quick to evaluate and returns `False` in a large proportion of cases, so that we can often avoid needing to evaluate `complexCondition`.

This does not require any additional extensions or special support in Haxl. But we also noticed that sometimes there is a pair of conditions where neither is obviously faster than the other, yet we would still like to benefit from bailing out early when the answer is known. Therefore, Haxl contains two more conditional operators `pOr` and `pAnd` for "parallel OR" and "parallel AND":

```
pOr, pAnd :: Haxl Bool -> Haxl Bool -> Haxl Bool
```

These have the behaviour that: (i) both arguments are evaluated in parallel; (ii) the computation is aborted as soon as the answer is known, even if the other argument is still being evaluated.

Data-fetches are not observable effects, so the parallelism is not observable to the programmer (this is the property that Haxl relies on for the soundness of its parallel `Applicative` instance). However, `pOr` and `pAnd` are non-deterministic with respect to exceptions: if an exception is thrown by either side, it will be thrown by the computation as a whole immediately without waiting for the other side to complete. One could imagine an alternative implementation which waits for the completion of the other argument when an exception is raised; this would be deterministic, but would be less efficient in the case of exceptions.

It should come as no surprise that `pOr` and `pAnd` can be implemented using `select`, indeed `pOr = (<||>)` and `pAnd = (<&&>)` from Fig. 3. A `Selective` instance in terms of the Haxl implementation from Marlow et al. [2014] is given in Fig. 6. For the purposes of the presentation here we have renamed `Fetch` to `Haxl` and added support for exceptions in the form of the `Throw` cases.

There is one wrinkle with implementing `pOr` and `pAnd` in terms of `select`. Ideally, `pOr` and `pAnd` would be symmetric: just as we can abort the right argument if the left argument determines the answer, we should be able to abort the left argument in the same way. Yet `select` is inherently left-biased: it requires that all the effects of the first argument are performed. In §6.2 we consider an alternative combinator related to `select` that allows this kind of symmetry to be expressed.

We have prototyped an implementation of Haxl with the `Selective Haxl` instance, which allowed us to reuse generic selective combinators `<||>`, `<&&>`, `anyS` and `allS` instead of providing custom implementations for conditional operators operators `pOr` and `pAnd` and their generalisations on lists. This case study highlights the fact that selective functors are useful not only in the static context, but in the dynamic context too, by allowing us to benefit from speculative execution.

4.1 Results

We mentioned above that `pOr` and `pAnd` are effective when the relative size of the conditional computations is unknown, so evaluating them in parallel with early exit is an effective alternative to either sequencing them manually (with **Monad**) or evaluating them in parallel to completion (with **Applicative**). This argument becomes even more compelling as the set of conditions to evaluate grows: imagine trying to efficiently sequence a set of ten or more conditions, and then repeating the exercise every time the set changes.

For this reason, in Haxl we found that list operations built on top of `pOr` and `pAnd`, which in this paper we call `andS` and `allS` (see Fig. 3), offer an important balance between performance and maintainability that is not provided by the **Applicative** or **Monad**-based combinators.

One could construct examples to demonstrate arbitrarily large performance gains from using `pOr` and `pAnd`, however that would not be particularly useful. Perhaps more useful would be a real-world measurement showing how much performance was improved in an actual application — but again, the value of that would depend to a large extent on how the application uses `pOr` and `pAnd`, and unfortunately the application code in our case is proprietary. Therefore instead we offer this anecdote: we first introduced a use of `pOr` to solve some performance issues in a complex production workload where we had long chains of conditionals that were difficult to optimise by hand, and `pOr` resulted in significant performance improvements.

5 FREE SELECTIVE FUNCTORS

The idea of describing effectful computations using *free constructions*, such as *free* [Swierstra 2008] and *freer monads* [Kiselyov and Ishii 2015] and *free applicative functors* [Capriotti and Kaposi 2014] is well-studied in the functional programming community. Free constructions allow us to focus on the internal aspects of the effect under consideration and receive the desired applicative or monadic computation structure *for free*, i.e. without the need to define custom instances or prove laws.

In this section we apply this idea to selective functors. We present a free construction for *rigid* selective functors (§5.1), and demonstrate it on two examples in §5.2 and §5.3.

5.1 Free construction

In the free structures methodology, the essence of an effect is captured by a data type that encodes the "commands" which the effect provides, acting as a deep embedding of the effect's interface. This data type needs only have enough structure to be a **Functor**. The purpose of a free construction is then to build a richer structure on top of this *base functor*, which would have the desired instances, in our case **Applicative** and **Selective**. In this section we will denote the base functor by `f`.

As we remarked in §2.3, rigid selective functors have a particularly simple normal form thanks to the additional law `(<*>) = apS`, which tells us that the apply operator `<*>` is redundant and can be implemented via the selective interface. This normal form has the following linear structure:

```
pure x <*? fa <*? fb <*? ... <*? fy
  where
    x  :: Either a (Either b (Either c (... z)))
    fa :: f (a ->   Either b (Either c (... z)))
    fb :: f (b ->             Either c (... z))
    ...
    fy :: f (y ->                       z)
```

In words, any rigid selective computation can be rewritten as a left-associated sequence of select operators, where the initial pure value `x` comes from a large sum type (comprising alternatives `a` to `z` in the above snippet), and each effect in the sequence of select operators eliminates one of the

```haskell
data Select f a where
    Pure   :: a -> Select f a
    Select :: Select f (Either a b) -> f (a -> b) -> Select f b

instance Functor f => Functor (Select f) where
    fmap f (Pure a)     = Pure (f a)
    fmap f (Select x y) = Select (fmap f <$> x) (fmap f <$> y) -- Free theorem from Fig. 4

instance Functor f => Applicative (Select f) where
    pure  = Pure
    (<*>) = apS -- Law of rigid selective functors

instance Functor f => Selective (Select f) where
    select x (Pure y)     = either y id <$> x -- Generalised identity
    select x (Select y z) = Select (select (f <$> x) (g <$> y)) (h <$> z) -- Associativity
      where
        f x = Right <$> x
        g y = \a -> bimap (,a) ($a) y
        h z = uncurry z

-- Lift a base functor into Select
liftSelect :: Functor f => f a -> Select f a
liftSelect f = Select (Pure (Left ())) (const <$> f)

-- Interpret a free selective structure given a natural transformation from f to g
runSelect :: Selective g => (forall x. f x -> g x) -> Select f a -> g a
runSelect _ (Pure a)     = pure a
runSelect t (Select x y) = select (runSelect t x) (t y)

-- Extract the resulting value from a pure selective computation
getPure :: Select f a -> Maybe a
getPure = runSelect (const Nothing)

-- Extract all possible effects from a selective computation
getEffects :: Functor f => Select f a -> [f ()]
getEffects = getOver . runSelect (Over . pure . void)
```

Fig. 7. Free rigid selective functors (various performance improvements omitted for clarity).

alternatives, in order, until only one remains (namely, `z`). Interestingly, there is no right-associative version of the normal form, because the associativity law can only be used to re-associate a given expression to the left (§2.3). To apply it in reverse, we would need a way to "factor out" the reshaping functions `f`, `g` and `h` from the base functor. This is different from applicative functors, which have two normal forms corresponding to left and right re-association [Capriotti and Kaposi 2014].

Fig. 7 gives an encoding of this normal form in Haskell. The free data type **`Select`** has two constructors: **`Pure`**, to embed pure values in a selective computation, and **`Select`** to build type-aligned sequences of base functor effects, by appending new effects to the right end of the sequence. The sequence can only be started with the **`Pure`** constructor, thus enforcing the normal form. Implementation of instances relies on the laws presented in §2.3: generalised identity, associativity, as well as one of the free theorems. We do not need the distributivity law, since it is subsumed by the law of rigid selective functors `(<*>) = apS`, which is used directly in the **`Applicative`** instance.

Effects from the base functor can be embedded in the free construction using the helper function `liftSelect`. To interpret a free selective computation **`Select`** `f a` in a selective functor `g`, one needs to provide a *natural transformation* from `f` to `g` to the function `runSelect`, which traverses the sequence of effects converting them to `g` and composing the results using `g`'s select operator.

For example, `getPure` reinterprets a given free computation in the selective functor `g = Maybe` using the natural transformation `const Nothing`, which leaves the `Pure` head of the sequence as is, but turns any subsequent effect into `Nothing`. Similarly, `getEffects` records all effects by stashing them in the selective functor `Over`, which are subsequently extracted from it by `getOver`.

We do not present a free construction for non-rigid selective functors; while useful, it is more complex and not particularly enlightening.

5.2 Ping-pong, freely

To illustrate the usage of the free selective functors on a simple example, we implement the classic Teletype DSL [Swierstra 2008] comprising two commands: *reading* a string form the input stream and *writing* a string to the output stream. The base functor has two corresponding constructors:

```
data Teletype a = Read (String -> a) | Write String a deriving Functor
```

For convenience, we can provide the following functions that embed the commands into the free selective construction, mimicking Haskell's `IO` API:

```
getLine :: Select Teletype String
getLine = liftSelect (Read id)

putStrLn :: String -> Select Teletype ()
putStrLn s = liftSelect (Write s ())
```

We can now reimplement the `pingPongS` example from §1 in terms of the free selective construction simply by adjusting the type signature. Note that the `whenS` combinator comes for free.

```
pingPongS :: Select Teletype ()
pingPongS = whenS (fmap (=="ping") getLine) (putStrLn "pong")
```

By embedding `pingPongS` into the free construction, we gain access to the static analysis machinery:

```
λ> getEffects pingPongS
[Read,Write "pong"]
```

The `getEffects` function (Fig. 7) returns a list of all effects of a free selective computation. In the case of the `Teletype` functor, we get a list of all commands that a computation might execute.

We can interpret `Teletype` programs in any other selective functor using the `runSelect` function by providing a natural transformation `forall x. Teletype x -> g x`, which assigns an interpretation to `Teletype` commands in terms of `g`. A good example of such transformation would be an interpretation in the `IO` monad, which allows us to execute our `pingPongS` program:

```
toIO :: Teletype a -> IO a
toIO (Read f)    = f <$> Prelude.getLine
toIO (Write s a) = a <$  Prelude.putStrLn s

λ> runSelect toIO pingPongS
hello
λ> runSelect toIO pingPongS
ping
pong
```

Note that while we can write simple programs like `pingPongS` using the selective interface, we are fundamentally limited in what we can express compared to the much more powerful monadic interface. As an example, consider this simple greeting program:

```
greeting = getLine >>= \name -> putStrLn ("Hello " ++ name)
```

Programs like this cannot be expressed in our simple Teletype DSL. Even if we had `bindS` for strings (§6.1), it would be useless for static analysis because it would have to report effects `Write name` for *all possible strings* `name`! Nevertheless, limitations of the selective interface can sometimes be worked around by using more sophisticated base functors, as we show in §5.3.

5.3 Analysis and simulation of processor instructions

To demonstrate the free construction on a more interesting example, we apply it to analysis and simulation of a hypothetical instruction set architecture (ISA)[5]. By expressing the ISA semantics in our free construction with an unusual base functor, we will be able to build tools both for static data flow analysis and simulation of programs with branching.

5.3.1 ISA semantics. To work around some of the aforementioned limitations of the selective interface (namely, the lack of the bind operator), we represent the semantics of instructions thus:

```
type ISA a = Select RW a

data RW a = Read  Key                (Value -> a)
          | Write Key (ISA Value) (Value -> a) deriving Functor
```

The `RW` (pronounced "read-write") base functor encodes the effect of a mutable key-value store comprising two commands: (i) we need the ability to *read* a value associated with a key from the store, and (ii) given a computation which produces a value, *write* its result into the store. Think of `Value` as a machine word, and `Key` as an ISA memory location (a register, a memory cell, or a processor flag). The base commands are similar to Teletype, with one key difference: the `Write` constructor takes `ISA Value`, i.e. a computation producing a value instead of just plain `Value`.

This exact structure of the definition is required for accommodating a pattern that occurs frequently in instruction semantics: often we read a value from a register or a memory cell, do something with it, and then write it somewhere else. If `Write` required the second argument to be a pure value, as in Teletype, we would not be able to express the desired pattern without resorting to the monadic interface. Additionally, we want the `Write` command to not just write the value and return `()`, but to *give the just written value back*, so it can be used in the rest of the computation; such generosity of the `Write` command will be useful for avoiding duplicating data dependencies.

We introduce two convenience combinators, which lift the data constructors of the `RW` data type into the free selective, thus making them directly usable in the definitions of instruction semantics:

```
read :: Key -> ISA Value
read k = liftSelect (Read k id)

write :: Key -> ISA Value -> ISA Value
write k fv = fv *> liftSelect (Write k fv id)
```

While the `read` combinator is exactly the lifted `Read` data constructor, the `write`'s implementation deviates from the trivial lifting of the `Write` data constructor and *evaluates its second argument*, thus executing the associated effects.

5.3.2 Example 1. Addition. To get acquainted with the introduced vocabulary, we start by describing the semantics for the addition instruction, which reads the summands from a register and a memory cell, adds them, writes the result back into the same register, and also updates the state of the `Zero` flag to indicate if the result is zero.

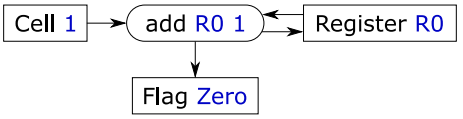
[5]Incidentally, this was the original motivation for selective functors. While describing the formal semantics of instructions of a real processor, we needed a statically analysable `ifS` for the purpose of symbolic program verification, which eventually led us to `select`. We use a hypothetical ISA in this section instead of the real one, because of the complexity of the latter.

```
add :: Register -> Address -> ISA Value
add reg addr = let arg1     = read (Reg reg)
                   arg2     = read (Cell addr)
                   result   = (+)   <$> arg1 <*> arg2
                   isZero   = (==0) <$> write (Reg reg) result
               in write (Flag Zero) (bool 0 1 <$> isZero)
```

Here, we `read` the summands `arg1` and `arg2` from the two specified locations and calculate the `result` of addition by lifting `(+)` into the free selective functor using applicative combinators. We then calculate the value of the `Zero` flag in a similar way, but here we exploit the fact that the `write` combinator returns the value it has just written, thus we can reuse the `result` without recalculating it from scratch and triggering the associated effects again.

By analysing the free semantics of the `add` instruction, we can obtain the list of all its effects, in the order they appear in the computation. We also visualise the effects as a data flow graph, where data locations are shown as rectangles, instructions as rounded rectangles, and reads/writes as arcs.

```
λ> getEffectsISA (add R0 1)
[Read R0,Read 1,Write R0,Write Zero]
```



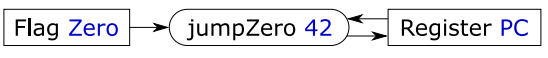
The semantics of the addition instruction has only used applicative combinators and we thus could have analysed it statically using free applicative functors. However, there are important instructions whose semantics cannot be expressed in terms of the `Applicative` interface, and this is where the presented free selective construction becomes irreplaceable.

5.3.3 Example 2. Conditional jump. Selective functors introduce limited dependencies between effectful computations, giving as enough power to express the semantics of branching instructions, which modify the program counter by a given offset if a certain condition holds. Consider the following instruction that performs a jump if the result of the previous operation was zero.

```
jumpZero :: Value -> ISA ()
jumpZero offset = let zeroSet  = (==1) <$> read (Flag Zero)
                      modifyPC = void $ write PC ((+offset) <$> read PC)
                  in whenS zeroSet modifyPC
```

Here we use the `whenS` combinator to modify the program counter only if the `Zero` flag is set. By implementing `jumpZero` in terms of the selective interface, we achieve both the ability to implement an adequate simulator for branching programs and perform their static analysis:

```
λ> getEffectsISA (jumpZero 42)
[Read Zero,Read PC,Write PC]
```



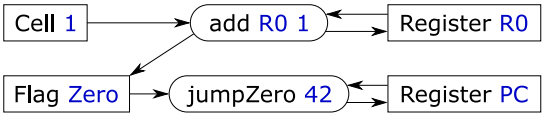
Since the analysis is static, the resulting list of effects and the corresponding data flow graph are over-approximated and show all effects that can possibly happen in the computation.

5.3.4 Blocks of instructions. Once we have implemented the semantics for a desired subset of an ISA, we can describe the semantics of sequences, or *blocks*, of instructions by simply composing the semantics of individual instructions using the applicative sequencing operator `(*>)`:

```
addAndJump :: ISA Value
addAndJump = add R0 1 *> jumpZero 42
```

We can analyse such compound computations in the same way as we analyse individual instructions:

```
λ> getEffectsISA addAndJump
[Read R0,Read 1,Write R0,Write Zero
,Read Zero,Read PC,Write PC]
```



5.3.5 Simulation. To implement an ISA simulator, we follow the same path as in the `pingPongS` example and the **`IO`** monad earlier in §5.2. We need a natural transformation from the base functor **`RW`** to an appropriate target functor, e.g. an instance of **`MonadState ISAState`**, where **`ISAState`** represents the state of all registers, memory cells and flags. For brevity, we present only one part of such a transformation, which assigns an interpretation to reading and writing of register keys **`Reg`**:

```
toState :: MonadState ISAState m => RW a -> m a
toState (Read k t) = t <$> case k of
    Reg r -> (Map.! r) <$> gets registers -- Look up r in the registers field
    ...
toState (Write k fv t) = case k of
    Reg r -> do v <- runSelect toState fv -- Evaluate the Write's argument first
                let step s = Map.insert r v (registers s)
                state $ \s -> (t v, s { registers = step s })
    ...
```

To read a register, we simply lift the lookup function **`Map`**`.!` to the corresponding field of the **`ISAState`**. To write an effectful value `fv` into a register, we need to evaluate it first; hence we call the `runSelect` function, supplying it the natural transformation `toState`, recursively, thus performing the effects of `fv`. We then adjust the register bank with the new value and return it.

The natural transformation `toState` gives interpretation to individual **`Read`** and **`Write`** commands, and now this interpretation can be extended to any **`ISA`** computation by plugging it into a `runSelect` call, as has already been done once in the implementation of `toState` itself:

```
runProgram :: ISA a -> ISAState -> (a, ISAState)
runProgram p = runState (runSelect toState p)
```

5.3.6 Restrictions. The free selective construction in combination with the base functor **`RW`** provides an abstraction capable of expressing the semantics of arithmetic, load/store and branching instructions. However, one should remember that selective functors still lack the full expressive power of the monadic interface and are unable to accommodate an important class of instructions, specifically those that use the *memory-indirect addressing mode*. If we had a **`Monad ISA`** instance, we could give the following semantics to the memory-indirect load instruction `loadMI`:

```
loadMI :: Register -> Address -> ISA Value
loadMI reg addr = read (Cell addr) >>= \x -> write (Reg reg) (read (Cell x))
```

Here, we read from a memory cell `addr`, then use the monadic bind operator to extract a value `x` from the result, and use it in a subsequent memory read *as address*. Although this semantics is, in principle, implementable using the selective `bindS` combinator (see §6.1), it is not very useful in practice since static analysis would record possible access to *all memory cells* `x`, and there are too many of them (typically, a large power of two). Furthermore, the execution of the resulting semantics would be terribly slow, since it would also follow the same linear exploration of the memory address space (although see §6.1 for a possible solution of the performance issue).

6 ALTERNATIVE FORMULATIONS FOR SELECTIVE FUNCTORS

This section discusses alternative versions of the `Selective` type class, which are based on multi-way generalisations of the `select` operator (§6.1), as well as on making it more symmetric (§6.2). Both of these ideas can be readily integrated into the presented definition of the `Selective` type class by extending it with new methods and adding new laws that ensure that the new methods interact with `select` in an appropriate manner. This is common in standard Haskell libraries, where type classes `Applicative` and `Monad` include methods like `*>` and `>>` for performance reasons.

Another alternative, which is worth a remark, is to simply move `select` to the `Applicative` type class, with the default implementation `select = selectA`. While this works for the purposes discussed in this paper, it would make it harder to reason about code with the `Applicative f` constraint, since the `select` method makes it possible for effects to depend on values; declaring such a significant ability by the `Selective f` constraint is arguably a more prudent approach.

6.1 Multiway select operators

As mentioned in §2, `branch` is a strong contender to be the main method of the `Selective` type class: it is parametric and all selective combinators, including `select` itself, can be derived from it:

```
branch :: Selective f => f (Either a b) -> f (a -> c) -> f (b -> c) -> f c

selectB :: Selective f => f (Either a b) -> f (a -> b) -> f b
selectB x y = branch x y (pure id)
```

While we prefer `select` for its simplicity, `branch` does provide an interesting advantage in the context of static analysis. Specifically, it makes it statically apparent that the two branches are *mutually exclusive*. When `branch` is "desugared" into a sequence of two `select` operations, the information about the mutual exclusion between the two branches is lost, which rules out some static analysis scenarios. For example, it may be useful to know that in our build systems example in §3.2 we never depend on both lib.c and lib.ml.

Another point in favour of `branch` is performance: the `select`-based implementation of the `ifS` combinator checks for the `Left` and `Right` cases in sequence, instead of directly jumping to the correct case, so a `branch`-based implementation would be more efficient. Furthermore, N-way generalisations of `select` are possible, although the design space here is quite large. As an example, one might consider adding `bindS` to the `Selective` type class, i.e. a special case of the monadic bind operator that is applicable only to enumerable types:

```
bindS :: Selective f => (Bounded a, Enum a, Eq a) => f a -> (a -> f b) -> f b
```

The default implementation could be based on sequentially checking for every possible value using `select`, but monadic instances would supply a much faster implementation, namely `bindS = (>>=)`. This would allow static analysis instances to record all possible cases, without incurring the $O(N)$ slowdown during the execution of an N-way branch.

Interestingly, adding the ability to branch on infinite number of cases makes selective functors equivalent to monads, although it is worth pointing out that static analysis of such infinitely-branching selective functors might take infinite time too.

Investigation of the design space for "multiway selective functors", and exploiting them for efficient translation of Haskell's do-notation into selective combinators in the spirit of the ApplicativeDo extension [Marlow et al. 2016] is left for future work. For now, we believe that adding `branch` and/or an equivalent of `bindS` to the `Selective` type class would be beneficial for performance-sensitive applications.

6.2 Symmetric select operators

In this section we address the asymmetry of `select` that we remarked on in §4. The asymmetry can be seen in the fact that the first argument of `select` must always be executed, while the second argument may sometimes be skipped. Consider the following more symmetric alternative:

```
biselect :: Selective f => f (Either a b) -> f (Either a c) -> f (Either a (b,c))
```

This definition is pleasantly symmetric: if either of the arguments yields a `Left a` value, the other argument *may be skipped* since the result must be a `Left a` too, by parametricity. On the other hand, if one of the arguments yields a `Right` value, then the other argument *must be executed* in order to either get an `a` or the other half of the resulting pair. As an added bonus, the rather obscure associativity law from §2.3 looks much more natural for `(?*?) = biselect`:

```
x ?*? (y ?*? z) = second assoc <$> ((x ?*? y) ?*? z)
  where
    assoc ((a, b), c) = (a, (b, c))
```

While beautiful, we found `biselect` to be a bit more awkward to work with than `select`, and also more subtle when the order in which the arguments are executed is not fixed. So far we have identified only one example where the symmetry of `biselect` is beneficial: speculative execution of parallel OR and AND combinators — see the Haxl case study §4. To support such use-cases it is possible to add `biselect` to the `Selective` type class with the following default implementation:

```
biselect :: Selective f => f (Either a b) -> f (Either a c) -> f (Either a (b,c))
biselect x y = select ((fmap Left . swap) <$> x) ((\e a -> fmap (a,) e) <$> y)
  where
    swap = either Right Left -- Swap Left and Right
```

This implementation breaks the symmetry, which may be acceptable for most instances of selective functors, but instances like `Haxl` would override it in order to gain additional performance benefits. Note that the selective combinators like `<||>` would need to be redefined via `biselect` in order to take advantage of the symmetry.

From the theoretical viewpoint, the type signature of `biselect` makes it more apparent that a selective functor `f` is a composition of an applicative functor `f` and the `Either` monad.

7 RELATED WORK

Composing effectful computations is a rich research area and there is a vast body of related work. We build on the fundamental notions of applicative functors [McBride and Paterson 2008] and monads [Moggi 1991; Wadler 1995], but these notions are not isolated: the space between them is inhabited by *arrows* [Hughes 2000] and *generalised arrows* [Megacz 2011], which we discuss in §7.1.

The idea of extending the `Applicative` interface to gain more expressive power is of course not new, and has been particularly popular in the area of *applicative parser combinators*. In particular, Swierstra and Duponcheel [1996] used an extended version of applicative functors that was later abstracted into the `Alternative` type class, which is discussed in §7.2. Other extensions include combinators for *observable recursion*, e.g. see [Devriese and Piessens 2012; Devriese et al. 2013]. Combining the latter with selective functors would give us the ability to statically analyse effectful computations with cycles.

Our free construction for rigid selective functors (§5) is inspired by the works on free applicative functors [Capriotti and Kaposi 2014], free monads [Swierstra 2008], and insightful blog posts by Fancher [2016, 2017]. *Batching and remote execution* of effectful computations [Gill et al. 2015] can be greatly simplified by using free applicative functors, as demonstrated by Gibbons [2016], and we believe that free selective functors uncover new opportunities in this area.

7.1 Arrows and profunctors

Arrows, introduced by Hughes [2000], generalise monads by making the *input* of a computation explicit. Rather than giving the type `f a` to an effectful computation that yields a value of type `a`, as we have done in this paper so far, arrows give the type `a i o` to an effectful computation that takes values of type `i` as input and yields values of type `o` as output. There is a rich *arrow hierarchy* of type classes, each providing a new ability, where **`ArrowChoice`** is particularly relevant for us:

```
class Category a                -- Identity arrow, sequential arrow composition
class Category a => Arrow       a -- Pure arrows, parallel arrow composition
class Arrow    a => ArrowChoice a -- Arrows with choice
class Arrow    a => ArrowApply  a -- Arrows that take arrows as input
class Arrow    a => ArrowLoop   a -- Arrows with loops
```

The relationships between applicative functors, monads and arrows have been studied in depth. It is known, e.g. see Lindley et al. [2011] and Rivas and Jaskelioff [2017], that applicative functors correspond to so called *static arrows*, for which there is an isomorphism between `a () (i -> o)` and `a i o`. The standard module Control.Arrow therefore provides the following definitions:

```
newtype ArrowMonad a b = ArrowMonad (a () b) -- See Control.Arrow

instance Arrow       a => Functor     (ArrowMonad a)
instance Arrow       a => Applicative (ArrowMonad a)
instance ArrowChoice a => ...                          -- Missing?!
instance ArrowApply  a => Monad       (ArrowMonad a)
```

Selective functors provide the missing counterpart for **`ArrowChoice`** in the *functor hierarchy*, as demonstrated by the following instance:

```
instance ArrowChoice a => Selective (ArrowMonad a) where
    select (ArrowMonad x) y = ArrowMonad $ x >>> (toArrow y ||| returnA)

toArrow :: Arrow a => ArrowMonad a (b -> c) -> a b c
toArrow (ArrowMonad f) = arr (\x -> ((), x)) >>> first f >>> arr (uncurry ($))
```

Here `toArrow` witnesses one half of the aforementioned isomorphism between `a () (i -> o)` and `a i o`. The obtained **`Selective`** instance is lawful thanks to the **`ArrowChoice`** laws.

Arrows are more general and powerful than selective functors. We could have used arrows to solve our static analysis and speculative execution examples, and not just in theory — Dune is a great example of successful application of arrows in practice. However, introducing arrows to an existing codebase built around applicative functors and monads, such as Haxl, would require pervasive changes to the whole abstraction stack, as well as rewriting all existing Haxl user code in the arrow notation [Paterson 2001]. Needless to say, introduction of selective functors to Haxl is a much easier task, which we have accomplished by adding 13 lines of new code (the definition of the **`Selective Haxl`** instance in Fig. 6), and removing 26 lines of code corresponding to similarly-sized definitions of `pOr` and `pAnd`, reusing the selective combinators `<||>` and `<&&>` instead.

Profunctors is an abstraction closely related to arrows; see [Pickering et al. 2017] for a good overview of profunctors in the context of modular data accessors, or *lenses*. Similarly to **`ArrowChoice`**, so-called *Cocartesian profunctors* are counterparts of selective functors in the *profunctor hierarchy*.

Establishing a formal correspondence between **`ArrowChoice`**, Cocartesian profunctors, and selective functors is beyond the scope of this paper and is left for future research.

7.2 Parsers and `Alternative` type class

`Alternative` is a type class originally motivated by non-monadic parsers; see, for example, Swierstra and Duponcheel [1996], where the methods of the `Alternative` type class appear as part a bigger `Parsing` type class. In modern Haskell, `Alternative` is a subclass of `Applicative`:

```
class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a
```

The operator `<|>` allows us to naturally express *choice* in parsers. As an example, consider the task of parsing binary and hexadecimal numbers, which are prefixed with `"0b"` and `"0x"`, respectively. Following the classic parser combinator approach [Hutton and Meijer 1998], let us assume the existence of the following parsers:

```
sat    :: (Char -> Bool) -> Parser Char   -- Parse a specified character
string :: String         -> Parser String -- Parse a string literal
bin    ::                   Parser Int    -- Parse a binary-encoded number
hex    ::                   Parser Int    -- Parse a hexadecimal-encoded number
```

Now the desired parser can be obtained as a choice between parsers for binary and hexadecimal numbers, each augmented with the prefix-parsing part:

```
numberA :: Parser Int
numberA = (string "0b" *> bin) <|> (string "0x" *> hex)
```

When parsing `"0x7E3"`, the first parser fails (due to the prefix mismatch), but the second one succeeds. Note that parsing of the leading `"0"` can be factored out into a separate parser `string "0"` to avoid backtracking.

Selective functors also allow us to implement the desired parser, and arguably in a more direct style that does not involve trying one parser after another:

```
numberS :: Parser Int
numberS = string "0" *> ifS (('b'==) <$> sat (`elem` "bx")) bin hex
```

Here we first parse the leading `"0"`, then the second character of the prefix, failing if it is neither `"b"` nor `"x"`, and finally select an appropriate subsequent parser using `ifS`. Note that we can move the parser `string "0"` in and out of the condition `ifS` thanks to the interchange law (§2.3).

Investigation of the relationship between `Alternative` and `Selective` type classes, as well as application of selective functors to parsers is an interesting research opportunity.

8 CONCLUSIONS

We have introduced selective functors, an abstraction between applicative functors and monads. Like applicative functors, selective functors require all effects to be known statically, before the execution starts. Like monads, selective functors allow for effects to depend on values of earlier effects but in a limited way: it is possible to skip some of the effects, but not create new ones. In this sense selective functors allow you to describe computations that are very much like hardware circuits: statically fixed, yet dynamically reconfigurable.

We have demonstrated usefulness of the new abstraction on several examples, and hope that the reader will find it useful in their next project too.

ACKNOWLEDGMENTS

We are very grateful to everyone who contributed by participating in numerous discussions and providing feedback on earlier drafts: Arseniy Alekseyev, Baldur Blöndal, Ulan Degenbaev, Dominique Devriese, Gábor Lehel, Neil Mitchell, Daniel Peebles, Artem Pelenitsyn, Simon Peyton Jones, Ivan Polyakov, Asad Saeeduddin, and reddit users Darwin226, dmwit and yakrar.

Andrey Mokhov's research is supported by a Royal Society Industry Fellowship IF160117 on the topic "Towards Cloud Build Systems with Dynamic Dependency Graphs".

REFERENCES

Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

Baldur Blöndal, Andres Löh, and Ryan Scott. 2018. Deriving Via. In *Proceedings of the 11th ACM Haskell Symposium (Haskell'18)*.

Paolo Capriotti and Ambrus Kaposi. 2014. Free applicative functors. *Proceedings 5th Workshop on Mathematically Structured Functional Programming* 153, 2–30.

Dominique Devriese and Frank Piessens. 2012. Finally tagless observable recursion for an abstract grammar model. *Journal of Functional Programming* 22, 6 (2012), 757–796.

Dominique Devriese, Ilya Sergey, Dave Clarke, and Frank Piessens. 2013. Fixing idioms: a recursion primitive for applicative DSLs. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*. ACM, 97–106.

Stephen Diehl. 2018. Boolean Blindness. (2018). http://dev.stephendiehl.com/hask/#boolean-blindness.

Will Fancher. 2016. More on Applicative Effects in Free Monads. (2016). https://elvishjerricco.github.io/2016/04/13/more-on-applicative-effects-in-free-monads.html.

Will Fancher. 2017. Profunctors, Arrows, & Static Analysis. (2017). https://elvishjerricco.github.io/2017/03/10/profunctors-arrows-and-static-analysis.html.

Jeremy Gibbons. 2016. Free delivery (functional pearl). In *ACM SIGPLAN Notices*, Vol. 51. ACM, 45–50.

Andy Gill, Neil Sculthorpe, Justin Dawson, Aleksander Eskilson, Andrew Farmer, Mark Grebe, Jeffrey Rosenbluth, Ryan Scott, and James Stanton. 2015. The remote monad design pattern. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 59–70.

John Hughes. 2000. Generalising monads to arrows. *Science of computer programming* 37, 1-3 (2000), 67–111.

Graham Hutton and Erik Meijer. 1998. Monadic parsing in Haskell. *Journal of functional programming* 8, 4 (1998), 437–444.

Jane Street. 2018. Dune: A composable build system. (2018). https://dune.build/.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. *SIGPLAN Not.* 50, 12 (Aug. 2015), 94–105. https://doi.org/10.1145/2887747.2804319

Sam Lindley, Philip Wadler, and Jeremy Yallop. 2011. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic notes in theoretical computer science* 229, 5 (2011), 97–117.

Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 325–337.

Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's do-notation into applicative operations. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 92–104.

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008), 1–13.

Adam Megacz. 2011. Hardware design with generalized arrows. In *International Symposium on Implementation and Application of Functional Languages*. Springer, 164–180.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build systems à la carte. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 79.

Ross Paterson. 2001. A new notation for arrows. In *ACM SIGPLAN Notices*, Vol. 36. ACM, 229–240.

Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. 2017. Profunctor optics: Modular data accessors. *arXiv preprint arXiv:1703.10857* (2017).

Exequiel Rivas and Mauro Jaskelioff. 2017. Notions of computation as monoids. *Journal of functional programming* 27 (2017).

S Doaitse Swierstra and Luc Duponcheel. 1996. Deterministic, error-correcting combinator parsers. In *International School on Advanced Functional Programming*. Springer, 184–207.

Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436.

The OPAM team. 2018. OCaml Package Manager. (2018). https://opam.ocaml.org/.

Philip Wadler. 1989. Theorems for free!. In *FPCA*, Vol. 89. 347–359.

P. Wadler. 1995. Monads for functional programming. In *Int'l School on Advanced Functional Programming*. Springer, 24–52.