Free Applicative Functors

Paolo Capriotti
University of Nottingham, United Kingdom
pvc@cs.nott.ac.uk

Ambrus Kaposi
University of Nottingham, United Kingdom
auk@cs.nott.ac.uk

Applicative functors [6] are a generalisation of monads. Both allow the expression of effectful computations into an otherwise pure language, like Haskell [5]. Applicative functors are to be preferred to monads when the structure of a computation is fixed *a priori*. That makes it possible to perform certain kinds of static analysis on applicative values. We define a notion of *free applicative functor*, prove that it satisfies the appropriate laws, and that the construction is left adjoint to a suitable forgetful functor. We show how free applicative functors can be used to implement embedded DSLs which can be statically analysed.

1 Introduction

Free monads in Haskell are a very well-known and practically used construction. Given any endofunctor `f`, the free monad on `f` is given by a simple inductive definition:

```
data Free f a
  = Return a
  | Free (f (Free f a))
```

The typical use case for this construction is creating embedded DSLs (see for example [10], where `Free` is called `Term`). In this context, the functor `f` is usually obtained as the coproduct of a number of functors representing "basic operations", and the resulting DSL is the minimal embedded language including those operations.
One problem of the free monad approach is that programs written in a monadic DSL are not amenable to static analysis. It is impossible to examine the structure of a monadic computation without executing it.
In this paper, we show how a similar "free construction" can be realized in the context of applicative functors. In particular, we make the following contributions:

- We give two definitions of *free applicative functor* in Haskell (section 2), and show that they are equivalent (section 5).
- We prove that our definition is correct, in the sense that it really is an applicative functor (section 6), and that it is "free" in a precise sense (section 7).
- We present a number of examples where the use of free applicative functors helps make the code more elegant, removes duplication or enables certain kinds of optimizations which are not possible when using free monads. We describe the differences between expressivity of DSLs using free applicatives and free monads (section 3).
- We compare our definition to other existing implementations of the same idea (section 10).

Submitted to:
MSFP 2014

Applicative functors can be regarded as monoids in the category of endofunctors with Day convolution (see for instance [3], example 3.2.2). There exists a general theory for constructing free monoids in monoidal categories [4], but in this paper we aim to describe the special case of applicative functors using a formalism that is accessible to an audience of Haskell programmers.
Familiarity with applicative functors is not required, although it is helpful to understand the motivation behind this work. We make use of category theoretical concepts to justify our definition, but the Haskell code we present can also stand on its own.
The proofs in this paper are carried out using equational reasoning in an informally defined total subset of Haskell. In sections 8 and 9 we will show how to interpret all our definitions and proofs in a general (locally presentable) cartesian closed category, such as the category of sets.

1.1 Applicative functors

Applicative functors (also called *idioms*) were first introduced in [6] as a generalisation of monads that provides a lighter notation for expressing monadic computations in an applicative style.
They have since been used in a variety of different applications, including efficient parsing (see section 1.4), regular expressions and bidirectional routing.
Applicative functors are defined by the following type class:

```
class Functor f ⇒ Applicative f where
  pure :: a → f a
  (<*>) :: f (a → b) → f a → f b
```

The idea is that a value of type `f a` represents an "effectful" computation returning a result of type `a`. The `pure` method creates a trivial computation without any effect, and `(<*>)` allows two computations to be sequenced, by applying a function returned by the first, to the value returned by the second.
Since every monad can be made into an applicative functor in a canonical way, the abundance of monads in the practice of Haskell programming naturally results in a significant number of practically useful applicative functors.
Applicatives not arising from monads, however, are not as widespread, probably because, although it is relatively easy to combine existing applicatives (see for example [7]), techniques to construct new ones have not been thoroughly explored so far.
In this paper we are going to define an applicative functor `FreeA f` for any Haskell functor `f`, thus providing a systematic way to create new applicatives, which can be used for a variety of applications.
The meaning of `FreeA f` will be clarified in section 7, but for the sake of the following examples, `FreeA f` can be thought of as the "simplest" applicative functor which can be built using `f`.

1.2 Example: option parsers

To illustrate how the free applicative construction can be used in practice, we take as a running example a parser for options of a command-line tool.
For simplicity, we will limit ourselves to an interface which can only accept options that take a single argument. We will use a double dash as a prefix for the option name.
For example, a tool to create a new user in a Unix system could be used as follows:

```
create_user --username john \
            --fullname "John Doe" \
            --id 1002
```

Our parser could be run over the argument list and it would return a record of the following type:

```
data User = User
  { username :: String
  , fullname :: String
  , id :: Int }
  deriving Show
```

Furthermore, given a parser, it should be possible to automatically produce a summary of all the options that it supports, to be presented to the user of the tool as documentation.
We can define a data structure representing a parser for an individual option, with a specified type, as a functor:

```
data Option a = Option
  { optName :: String
  , optDefault :: Maybe a
  , optReader :: String → Maybe a }
  deriving Functor
```

We now want to create a DSL based on the `Option` functor, which would allow us to combine options for different types into a single value representing the full parser. As stated in the introduction, a common way to create a DSL from a functor is to use free monads.
However, taking the free monad over the `Option` functor would not be very useful here. First of all, sequencing of options should be *independent*: later options should not depend on the value parsed by previous ones. Secondly, monads cannot be inspected without running them, so there is no way to obtain a summary of all options of a parser automatically.
What we really need is a way to construct a parser DSL in such a way that the values returned by the individual options can be combined using an `Applicative` interface. And that is exactly what `FreeA` will provide.
Thus, if we use `FreeA Option a` as our embedded DSL, we can interpret it as the type of a parser with an unspecified number of options, of possibly different types. When run, those options would be matched against the input command line, in an arbitrary order, and the resulting values will be eventually combined to obtain a final result of type `a`.
In our specific example, an expression to specify the command line option parser for `create_user` would look like this:

```
userP :: FreeA Option User
userP = User
  <$> one (Option "username" Nothing Just)
  <*> one (Option "fullname" (Just "") Just)
  <*> one (Option "id" Nothing readInt)
readInt :: String → Maybe Int
```

where we need a “generic smart constructor”:

```
one :: Option a → FreeA Option a
```

which lifts an option to a parser.

1.3 Example: web service client

One of the applications of free monads, exemplified in [10], is the definition of special-purpose monads, allowing to express computations which make use of a limited and well-defined subset of IO operations. Given the following functor:

```
data WebService a =
    GET  {url :: URL, params :: [String], result :: (String → a)}
  | POST {url :: URL, params :: [String], body :: String, cont :: a}
  deriving Functor
```

the free monad on WebService, once "smart constructors" are defined for the two basic operations of getting and posting, allows the definition of an application interacting with a web service with the same convenience as the IO monad.
For example, one can implement an operation which copies data from one server to another as follows:

```
copy :: URL → [String] → URL → [String] → Free WebService ()
copy srcURL srcPars dstURL dstPars = get srcURL srcPars >>= post dstURL dstPars
```

For some applications, we might need to have more control over the operations that are going to be executed when we eventually run the embedded program contained in a value of type Free WebService a. For example, a web service client application executing a large number of GET and POST operations might want to rate limit the number of requests to a particular server by putting delays between them, and, on the other hand, parallelise requests to different servers. Another useful feature would be to estimate the time it would take to execute an embedded Web Service application.
However, there is no way to achieve that using the free monad approach. In fact, it is not even possible to define a function like:

```
count :: Free WebService a → Int
```

which returns the total number of GET/POST operations performed by a value of type Free WebService a. To see why, consider the following example, which updates the email field in all the blog posts on a particular website:

```
updateEmails :: String → Free WebService ()
updateEmails newEmail = do
  entryURLs ← get "myblog.com" ["list_entries"]
  forM_ (words entryURLs) $ λentryURL →
    post entryURL ["updateEmail"] newEmail
```

Now, the number of POST operations performed by updateEmails is the same as the number of blog posts on myblog.com which cannot be determined by a pure function like count.
The FreeA construction, presented in this paper, represents a general solution for the problem of constructing embedded languages that allow the definition of functions performing static analysis on embedded programs, of which count :: FreeA WebService a → Int is a very simple example.

1.4 Example: applicative parsers

The idea that monads are “too flexible” has also been explored, again in the context of parsing, by Swierstra and Duponcheel [9], who showed how to improve both performance and error-reporting capabilities of an embedded language for grammars by giving up some of the expressivity of monads.
The basic principle is that, by weakening the monadic interface to that of an applicative functor (or, more precisely, an *alternative* functor), it becomes possible to perform enough static analysis to compute first sets for productions.
The approach followed in [9] is ad-hoc: an applicative functor is defined, which keeps track of first sets, and whether a parser accepts the empty string. This is combined with a traditional monadic parser, regarded as an applicative functor, using a generalized semi-direct product, as described in [7].
The question, then, is whether it is possible to express this construction in a general form, in such a way that, given a functor representing a notion of “parser” for an individual symbol in the input stream, applying the construction one would automatically get an Applicative functor, allowing such elementary parsers to be sequenced.
Free applicative functors can be used to that end. We start with a functor `f`, such that `f a` describes an elementary parser for individual elements of the input, returning values of type `a`. `FreeA f a` is then a parser which can be used on the full input, and combines all the outputs of the individual parsers out of which it is built, yielding a result of type `a`.
Unfortunately, applying this technique directly results in a strictly less expressive solution. In fact, since `FreeA f` is the simplest applicative over `f`, it is necessarily *just* and applicative, i.e. it cannot also have an `Alternative` instance, which in this case is essential.
The `Alternative` type class is defined as follows:

```
class Applicative f ⇒ Alternative f where
  empty :: f a
  (<|>) :: f a → f a → f a
```

An Alternative instance gives an applicative functor the structure of a monoid, with `empty` as the unit element, and `<|>` as the binary operation. In the case of parsers, `empty` matches the empty input string, while `<|>` is a choice operator between two parsers.
We discuss the issue of `Alternative` in more detail in section 11.

2 Definition of free applicative functors

To obtain a suitable definition for the free applicative functor generated by a functor `f`, we first pause to reflect on how one could naturally arrive at the definition of the `Applicative` class via an obvious generalisation of the notion of functor.
Given a functor `f`, the `fmap` method gives us a way to lift *unary* pure functions `a → b` to effectful functions `f a → f b`, but what about functions of arbitrary arity?
For example, given a value of type `a`, we can regard it as a nullary pure function, which we might want to lift to a value of type `f a`.
Similarly, given a binary function `h :: a → b → c`, it is quite reasonable to ask for a lifting of `h` to something of type `f a → f b → f c`.
The `Functor` instance alone cannot provide either of such liftings, nor any of the higher-arity liftings which we could define.

It is therefore natural to define a type class for generalised functors, able to lift functions of arbitrary arity:

```
class Functor f ⇒ MultiFunctor f where
  fmap₀ :: a → f a
  fmap₁ :: (a → b) → f a → f b
  fmap₁ = fmap
  fmap₂ :: (a → b → c) → f a → f b → f c
```

It is easy to see that a higher-arity $\texttt{fmap}_\texttt{n}$ can now be defined in terms of $\texttt{fmap}_2$. For example, for $\texttt{n} = 3$:

```
fmap₃ :: MultiFunctor f
      ⇒ (a → b → c → d)
      → f a → f b → f c → f d
fmap₃ h x y z = fmap₂ ($) (fmap₂ h x y) z
```

However, before trying to think of what the laws for such a type class ought to be, we can observe that `MultiFunctor` is actually none other than `Applicative` in disguise.
In fact, $\texttt{fmap}_0$ has exactly the same type as `pure`, and we can easily convert $\texttt{fmap}_2$ to `(<*>)` and vice versa:

```
g <*> x = fmap₂ ($) g x
fmap₂ h x y = fmap h x <*> y
```

The difference between `(<*>)` and $\texttt{fmap}_2$ is that `(<*>)` expects the first two arguments of $\texttt{fmap}_2$, of types `a → b → c` and `f a` respectively, to be combined in a single argument of type `f (b → c)`.
This can always be done with a single use of `fmap`, so, if we assume that `f` is a functor, `(<*>)` and $\texttt{fmap}_2$ are effectively equivalent.
Nevertheless, this roundabout way of arriving to the definition of `Applicative` shows that an applicative functor is just a functor that knows how to lift functions of arbitrary arities. An overloaded notation to express the application of $\texttt{fmap}_i$ for all i is defined in [6], where it is referred to as *idiom brackets*.
Given a pure function of arbitrary arity and effectful arguments:

$$\begin{aligned}
&\texttt{h} : \texttt{b}_1 \to \texttt{b}_2 \to \cdots \to \texttt{b}_n \to \texttt{a}\\
&\texttt{x}_1 : \texttt{f}\ \texttt{b}_1\\
&\texttt{x}_2 : \texttt{f}\ \texttt{b}_2\\
&\cdots\\
&\texttt{x}_n : \texttt{f}\ \texttt{b}_n
\end{aligned}$$

the idiom bracket notation is defined as:

$$[\![\, \texttt{h}\ \texttt{x}_1\ \texttt{x}_2 \cdots \texttt{x}_n \,]\!] = \texttt{pure}\ \texttt{h} \ \texttt{<*>}\ \texttt{x}_1 \ \texttt{<*>}\ \texttt{x}_2 \ \texttt{<*>} \cdots \texttt{<*>}\ \texttt{x}_n$$

We can build such an expression formally by using a `PureL` constructor corresponding to `pure` and a left-associative infix `(:*:)` constructor corresponding to `(<*>)`:

$$\texttt{PureL}\ \texttt{h}\ \texttt{:*:}\ \texttt{x}_1\ \texttt{:*:}\ \texttt{x}_2\ \texttt{:*:} \cdots \texttt{:*:}\ \texttt{x}_n$$

The corresponding inductive definition is:

```
data FreeAL f a
   = PureL a
   | ∀b.FreeAL f (b → a) :*: f b
infixl 4 :*:
```

The `MultiFunctor` typeclass, the idiom brackets and the `FreeAL` definition correspond to the left parenthesised canonical form[1] of expressions built with `pure` and `(<*>)`. Just as lists built with concatenation have two canonical forms (cons-list and snoc-list) we can also define a right-parenthesised canonical form for applicative functors — a pure value over which a sequence of effectful functions are applied:

```
x : b₁
h₁ : f (b₁ → b₂)
h₂ : f (b₂ → b₃)
...
hₙ : f (bₙ → a)
hₙ <*> (··· <*> (h₂ <*> (h₁ <*> pure x)) ···)
```

Replacing `pure` with a constructor `Pure` and `(<*>)` by a right-associative infix `(:$:)` constructor gives the following expression:

```
hₙ :$: ··· :$: h₂ :$: h₁ :$: Pure x
```

The corresponding inductive type:

```
data FreeA f a
   = Pure a
   | ∀b.f (b → a) :$: FreeA f b
infixr 4 :$:
```

`FreeAL` and `FreeA` are isomorphic (see section 5); we pick the right-parenthesised version as our official definition since it is simpler to define the `Functor` and `Applicative` instances:

```
instance Functor f ⇒ Functor (FreeA f) where
  fmap g (Pure x)  = Pure (g x)
  fmap g (h :$: x) = fmap (g ∘) h :$: x
```

The functor laws can be verified by structural induction, simply applying the definitions and using the functor laws for `f`.

```
instance Functor f ⇒ Applicative (FreeA f) where
  pure = Pure
  Pure g <*> y     = fmap g y
  (h :$: x) <*> y = fmap uncurry h :$:
                     ((,) <$> x <*> y)
```

[1]Sometimes called simplified form because it is not necessarily unique.

In the last clause of the `Applicative` instance, `h` has type `f (x → y → z)`, and we need to return a value of type `FreeA f z`. Since `(:$:)` only allows us to express applications of 1-argument "functions", we uncurry `h` to get a value of type `f ((x , y) → z)`, then we use `(<*>)` recursively (see section 8 for a justification of this recursive call) to pair `x` and `y` into a value of type `FreeA f (x , y)`, and finally use the `(:$:)` constructor to build the result. Note the analogy between the definition of `(<*>)` and `(++)` for lists.

3 Applications

3.1 Example: option parsers (continued)

By using our definition of free applicative, we can compose the command line option parser exactly as shown in section 1.2 in the definition of `userP`. The smart constructor `one` which lifts an option (a functor representing a basic operation of our embedded language) to a term in our language can now be implemented as follows:

```
one :: Option a → FreeA Option a
one opt = fmap const opt :$: Pure ()
```

A function which computes the global default value of a parser can also be defined:

```
parserDefault :: FreeA Option a → Maybe a
parserDefault (Pure x) = Just x
parserDefault (g :$: x) =
  optDefault g <*> parserDefault x
```

In section 7 we show that our definition is a free construction which gives us general ways to structure programs. Specifically, we are able to define a generic version of `one` which works for any functor. By exploiting the adjunction describing the free construction we are able to shorten the definition of `parserDefault`, define a function listing all possible options and a function parsing a list of command line arguments given in arbitrary order (section 7.1).

3.2 Example: web service client (continued)

In section 1.3 we showed an embedded DSL for web service clients based on free monads does not support certain kinds of static analysis.
However, we can now remedy this by using a free applicative, over the same functor `WebService`. In fact, the `count` function is now definable for `FreeA WebService a`. Moreover, this is not limited to this particular example: it is possible to define `count` for the free applicative over *any* functor.

```
count :: FreeA f a → Int
count (Pure _) = 0
count (_ :$: u) = 1 + count u
```

Static analysis of the embedded code now also allows decorating requests with parallelization instructions statically as well as rearranging requests to the same server.
Of course, the extra power comes at a cost. Namely, the expressivity of the corresponding embedded language is severely reduced.

Using `FreeA WebService`, all the URLs of the servers to which requests are sent must be known in advance, as well as the parameters and content of every request.
In particular, what one posts to a server cannot depend on what has been previously read from another server, so operations like `copy` cannot be implemented.

3.3 Summary of examples

Applicative functors are useful for describing certain kinds of effectful computations. The free applicative construct over a given functor specifying the "basic operations" of an embedded language gives rise to terms of the embedded DSL built by applicative operators. These terms are only capable of representing a certain kind of effectful computation which can be described best with the help of the left-parenthesised canonical form: a pure function applied to effectful arguments. The calculation of the arguments may involve effects but in the end the arguments are composed by a pure function, which means that the effects performed are fixed when specifying the applicative expression.
In the case of the option parser example `userP`, the pure function is given by the `User` constructor and the "basic operation" `Option` is defining an option. The effects performed depend on how an evaluator is defined over an expression of type `FreeA Option a` and the order of effects can depend on the implementation of the evaluator.
For example, if one defines an embedded language for querying a database, and constructs applicative expressions using `FreeA`, one might analyze the applicative expression and collect information on the individual database queries by defining functions similar to the `count` function in the web service example. Then, different, possibly expensive duplicate queries can be merged and performed at once instead of executing the effectful computations one by one. By restricting the expressivity of our language we gain freedom in defining how the evaluator works.
One might define parts of an expression in an embedded DSL using the usual free monad construction, other parts using `FreeA` and compose them by lifting the free applicative expression to the free monad using the following function:

```
liftA2M :: Functor f ⇒ FreeA f a → Free f a
liftA2M (Pure x) = Return x
liftA2M (h :$: x) = Free
  (fmap (λf → fmap f (liftA2M x)) h)
```

In the parts of the expression defined using the free monad construction, the order of effects is fixed and the effects performed can depend on the result of previous effectful computations, while the free applicative parts have a fixed structure with effects not depending on each other. The monadic parts of the computation can depend on the result of static analysis carried out over the applicative part:

```
test :: FreeA FileSystem Int → Free FileSystem ()
test op = do
  ...
  let n = count op    -- result of static analysis
  n′ ← liftA2M op     -- result of applicative computation
  max ← read "max"
  when (max ⩾ n + n′) $ write "/tmp/test" "blah"
  ...
```

The possibility of using the results of static analysis instead of the need of specifying them by hand (in our example, this would account to counting certain function calls in an expression by looking at the code) can make the program less redundant.

4 Parametricity

In order to prove anything about our free applicative construction, we need to make an important observation about its definition.
The `(:$:)` constructor is defined using an existential type `b`, and it is clear intuitively that there is no way, given a value of the form `g :$: x`, to make use of the type `b` hidden in it.
More specifically, any function on `FreeA f a` must be defined *polymorphically* over all possible types `b` which could be used for the existentially quantified variable in the definition of `(:$:)`.
To make this intuition precise, we appeal to the notion of *relational parametricity* [8] [11], which, specialised to the `(:$:)` constructor, implies that:

```
(:$:) :: ∀b.f (b → a) → (FreeA f b → FreeA f a)
```

is a natural transformation of contravariant functors. The two contravariant functors here could be defined, in Haskell, using a **newtype**:

```
newtype F1 f a x = F1 (f (x → a))
newtype F2 f a x = F2 (FreeA f x → FreeA f a)

instance Functor f ⇒ Contravariant (F1 f a) where
  contramap h (F1 g) = F1 $ fmap (∘h) g

instance Functor f ⇒ Contravariant (F2 f a) where
  contramap h (F2 g) = F2 $ g ∘ fmap h
```

The action of `F1` and `F2` on morphisms is defined in the obvious way. Note that here we make use of the fact that `FreeA f` is a functor.
Naturality of `(:$:)` means that, given types `x` and `y`, and a function `h : x → y`, the following holds:

$$\forall g :: f\ (y \to a), u :: \mathtt{FreeA}\ f\ x.$$
$$\mathtt{fmap}\ (\circ h)\ g \mathbin{:\$:} u \equiv g \mathbin{:\$:} \mathtt{fmap}\ h\ u \quad (1)$$

where we have unfolded the definitions of `contramap` for `F1` and `F2`, and removed the newtypes.

5 Isomorphism of the two definitions

In this section we show that the two definitions of free applicatives given in section 2 are isomorphic.
First of all, if `f` is a functor, `FreeAL f` is also a functor:

```
instance Functor f ⇒ Functor (FreeAL f) where
  fmap g (PureL x) = PureL (g x)
  fmap g (h :*: x)  = (fmap (g∘) h) :*: x
```

Again, the functor laws can be verified by a simple structural induction.

For the `(:*:)` constructor, a free theorem can be derived in a completely analogous way to deriving equation 1. This equation states that `(:*:)` is a natural transformation:

$$\forall h :: x \to y, g :: \texttt{FreeAL}\ f\ (y \to a), u :: f\ x\,.$$
$$\texttt{fmap}\ (\circ h)\ g \mathbin{:*:} u \equiv g \mathbin{:*:} \texttt{fmap}\ h\ u \qquad (2)$$

We define functions to convert between the two definitions:

```
r2l :: Functor f ⇒ FreeA f a → FreeAL f a
r2l (Pure x) = PureL x
r2l (h :$: x) = fmap (flip ($)) (r2l x) :*: h
l2r :: Functor f ⇒ FreeAL f a → FreeA f a
l2r (PureL x) = Pure x
l2r (h :*: x) = fmap (flip ($)) x :$: l2r h
```

We will also need the fact that `l2r` is a natural transformation:

$$\forall h :: x \to y, u :: \texttt{FreeAL}\ f\ x\,.$$
$$\texttt{l2r}\ (\texttt{fmap}\ h\ u) \equiv \texttt{fmap}\ h\ (\texttt{l2r}\ u) \qquad (3)$$

Proposition 1. `r2l` *is an isomorphism, the inverse of which is* `l2r`.

Proof. First we prove that $\forall u :: \texttt{FreeA}\ f\ a\,.\,\texttt{l2r}\ (\texttt{r2l}\ u) \equiv u$. We compute using equational reasoning with induction on `u`:

```
  l2r (r2l (Pure x))
≡ ⟨ definition of r2l ⟩
  l2r (PureL x)
≡ ⟨ definition of l2r ⟩
  Pure x
```

```
  l2r (r2l (h :$: x))
≡ ⟨ definition of r2l ⟩
  l2r (fmap (flip ($)) (r2l x) :*: h)
≡ ⟨ definition of l2r ⟩
  fmap (flip ($)) h :$:
  l2r (fmap (flip ($)) (r2l x))
≡ ⟨ equation 3 ⟩
  fmap (flip ($)) h :$:
  fmap (flip ($)) (l2r (r2l x))
≡ ⟨ inductive hypothesis ⟩
  fmap (flip ($)) h :$: fmap (flip ($)) x
≡ ⟨ equation 1 ⟩
  fmap (∘ (flip ($))) (fmap (flip ($)) h) :$: x
≡ ⟨ f is a functor ⟩
  fmap ((∘ (flip ($))) ∘ flip ($)) h :$: x
≡ ⟨ definition of flip and ($) ⟩
```

```
  fmap id h :$: x
≡ ⟨ f is a functor ⟩
  h :$: x
```

Next, we prove that $\forall$`u :: FreeAL f a . r2l (l2r u)` ≡ `u`. Again, we compute using equational reasoning with induction on `u`:

```
  r2l (l2r (PureL x))
≡ ⟨ definition of l2r ⟩
  r2l (Pure x)
≡ ⟨ definition of r2l ⟩
  PureL x

  r2l (l2r (h :*: x))
≡ ⟨ definition of l2r ⟩
  r2l (fmap (flip ($)) x :$: l2r h)
≡ ⟨ definition of r2l ⟩
  fmap (flip ($)) (r2l (l2r h)) :*: fmap (flip ($)) x
≡ ⟨ inductive hypothesis ⟩
  fmap (flip ($)) h :*: fmap (flip ($)) x
≡ ⟨ equation 2 ⟩
  fmap (∘ (flip ($))) (fmap (flip ($)) h) :*: x
≡ ⟨ FreeAL f is a functor ⟩
  fmap ((∘ (flip ($))) ∘ flip ($)) h :*: x
≡ ⟨ definition of flip and ($) ⟩
  fmap id h :*: x
≡ ⟨ FreeAL f is a functor ⟩
  h :*: x
```

□

In the next sections, we will prove that `FreeA` is a free applicative functor. Because of the isomorphism of the two definitions, these results will carry over to `FreeAL`.

6 Applicative laws

Following [6], the laws for an `Applicative` instance are:

$$\texttt{pure id <*> u} \equiv \texttt{u} \tag{4}$$

$$\texttt{pure (}\circ\texttt{) <*> u <*> v <*> x} \equiv \texttt{u <*> (v <*> x)} \tag{5}$$

$$\texttt{pure f <*> pure x} \equiv \texttt{pure (f x)} \tag{6}$$

$$\texttt{u <*> pure x} \equiv \texttt{pure (\$ x) <*> u} \tag{7}$$

We introduce a few abbreviations to help make the notation lighter:

```
uc = uncurry
pair x y = ( , ) <$> x <*> y
```

Lemma 1. *For all*

$$\begin{aligned} &\mathtt{u} :: \mathtt{y} \to \mathtt{z} \\ &\mathtt{v} :: \mathtt{FreeA}\ \mathtt{f}\ (\mathtt{x} \to \mathtt{y}) \\ &\mathtt{x} :: \mathtt{FreeA}\ \mathtt{f}\ \mathtt{x} \end{aligned}$$

the following equation holds:

$$\mathtt{fmap}\ \mathtt{u}\ (\mathtt{v} \mathbin{\texttt{<*>}} \mathtt{x}) \equiv \mathtt{fmap}\ (\mathtt{u}\circ)\ \mathtt{v} \mathbin{\texttt{<*>}} \mathtt{x}$$

Proof. We compute:

```
  fmap u (Pure v <*> x)
≡ ⟨ definition of (<*>) ⟩
  fmap u (fmap v x)
≡ ⟨ FreeA f is a functor ⟩
  fmap (u ∘ v) x
≡ ⟨ definition of (<*>) ⟩
  Pure (u ∘ v) <*> x
≡ ⟨ definition of fmap ⟩
  fmap (u ∘) (Pure v) <*> x
```

```
  fmap u ((g :$: y) <*> x)
≡ ⟨ definition of (<*>) ⟩
  fmap u (fmap uc g :$: pair y x)
≡ ⟨ definition of fmap ⟩
  fmap (u ∘) (fmap uc g) :$: pair y x
≡ ⟨ f is a functor ⟩
  fmap (λg → u ∘ uc g) g :$: pair y x
≡ ⟨ f is a functor ⟩
  fmap uc (fmap ((u ∘) ∘) g) :$: pair y x
≡ ⟨ definition of (<*>) ⟩
  (fmap ((u ∘) ∘) g :$: y) <*> x
≡ ⟨ definition of fmap ⟩
  fmap (u ∘) (g :$: y) <*> x
```

□

Lemma 2. *Property 5 holds for* `FreeA f`, *i.e. for all*

$$\begin{aligned} &\mathtt{u} :: \mathtt{FreeA}\ \mathtt{f}\ (\mathtt{y} \to \mathtt{z}) \\ &\mathtt{v} :: \mathtt{FreeA}\ \mathtt{f}\ (\mathtt{x} \to \mathtt{y}) \\ &\mathtt{x} :: \mathtt{FreeA}\ \mathtt{f}\ \mathtt{x}, \end{aligned}$$

$$\mathtt{pure}\ (\circ) \mathbin{\texttt{<*>}} \mathtt{u} \mathbin{\texttt{<*>}} \mathtt{v} \mathbin{\texttt{<*>}} \mathtt{x} \equiv \mathtt{u} \mathbin{\texttt{<*>}} (\mathtt{v} \mathbin{\texttt{<*>}} \mathtt{x})$$

Proof. Suppose first that $u = \mathtt{Pure}\ u_0$ for some $u_0 :: y \to z$:

```
  Pure (∘) <*> Pure u_0 <*> v <*> x
≡ ⟨ definition of (<*>) ⟩
  Pure (u_0 ∘) <*> v <*> x
≡ ⟨ definition of (<*>) ⟩
  fmap (u_0 ∘) v <*> x
≡ ⟨ lemma 1 ⟩
  fmap u_0 (v <*> x)
≡ ⟨ definition of (<*>) ⟩
  Pure u_0 <*> (v <*> x)
```

To tackle the case where $u = g\ \mathtt{:\$:}\ w$, for

$$g :: f\ (w \to y \to z)$$
$$w :: \mathtt{FreeA}\ f\ w,$$

we need to define a helper function

```
t :: ((w, x → y), x) → (w, y)
t ((w, v), x) = (w, v x)
```

and compute:

```
  pure (∘) <*> (g :$: w) <*> v <*> x
≡ ⟨ definition of pure and (<*>) ⟩
  (fmap ((∘) ∘) g :$: w) <*> v <*> x
≡ ⟨ definition of composition ⟩
  (fmap (λg w v → g w ∘ v) g :$: w) <*> v <*> x
≡ ⟨ definition of (<*>) ⟩
  (fmap uc (fmap (λg w v → g w ∘ v) g) :$: pair w v)
  <*> x
≡ ⟨ f is a functor and definition of uc ⟩
  (fmap (λg (w, v) → g w ∘ v) g :$: pair w v) <*> x
≡ ⟨ definition of (<*>) ⟩
  fmap uc (fmap (λg (w, v) → g w ∘ v) g) :$:
  pair (pair w v) x
≡ ⟨ f is a functor and definition of uc ⟩
  fmap (λg ((w, v), x) → g w (v x)) g :$:
  pair (pair w v) x
≡ ⟨ definition of uc and t ⟩
  fmap (λg → uc g ∘ t) g :$: pair (pair w v) x
≡ ⟨ f is a functor ⟩
  fmap (∘t) (fmap uc g) :$: pair (pair w v) x
≡ ⟨ equation 1 ⟩
  fmap uc g :$: fmap t (pair (pair w v) x)
≡ ⟨ lemma 1 (3 times) and FreeA f is a functor (3 times) ⟩
```

```
  fmap uc g :$: (pure (∘) <*> fmap (,) w <*> v <*> x)
≡ ⟨ induction hypothesis for fmap (,) w ⟩
  fmap uc g :$: (fmap (,) w <*> (v <*> x))
≡ ⟨ definition of (<*>) ⟩
  (g :$: w) <*> (v <*> x)
```

□

Lemma 3. *Property 7 holds for* `FreeA f`*, i.e. for all*

$$\begin{aligned}&\mathtt{u :: FreeA\ f\ (x \to y)}\\&\mathtt{x :: x},\end{aligned}$$

$$\mathtt{u <*> pure\ x \equiv pure\ (\$x) <*> u}$$

Proof. If `u` is of the form `Pure` u_0, then the conclusion follows immediately.
Let's assume, therefore, that `u = g :$: w`, for some `w :: w`, `g :: f (w → x → y)`, and that the lemma is true for structurally smaller values of `u`:

```
  (g :$: w) <*> pure x
≡ ⟨ definition of (<*>) ⟩
  fmap uc g :$: pair w (pure x)
≡ ⟨ definition of pair ⟩
  fmap uc g :$: (fmap (,) w <*> pure x)
≡ ⟨ induction hypothesis for fmap (,) w ⟩
  fmap uc g :$: (pure ($x) <*> fmap (,) w)
≡ ⟨ FreeA f is a functor ⟩
  fmap uc g :$: fmap (λw → (w, x)) w)
≡ ⟨ equation 1 ⟩
  fmap (λg w → g (w, x)) (fmap uc g) :$: w
≡ ⟨ f is a functor ⟩
  fmap (λg w → g w x) g :$: w
≡ ⟨ definition of fmap for FreeA f ⟩
  fmap ($x) (g :$: w)
≡ ⟨ definition of (<*>) ⟩
  pure ($x) <*> (g :$: w)
```

□

Proposition 2. `FreeA f` *is an applicative functor.*

Proof. Properties 4 and 6 are straightforward to verify using the fact that `FreeA f` is a functor, while properties 5 and 7 follow from lemmas 2 and 3 respectively. □

7 `FreeA` as a Left adjoint

We are now going to make the statement that `FreeA f` is the free applicative functor on `f` precise.
First of all, we will define a category $\mathscr{A}$ of applicative functors, and show that `FreeA` is a functor

$$\texttt{FreeA} : \mathscr{F} \to \mathscr{A},$$

where $\mathscr{F}$ is the category of endofunctors of `Hask`.
Saying that `FreeA f` is the free applicative on `f`, then, amounts to saying that `FreeA` is left adjoint to the forgetful functor $\mathscr{A} \to \mathscr{F}$.

Definition 1. *Let* `f` *and* `g` *be two applicative functors. An applicative natural transformation between* `f` *and* `g` *is a polymorphic function*

```
t :: ∀a.f a → g a
```

satisfying the following laws:

$$\texttt{t (pure x)} \equiv \texttt{pure x} \tag{8}$$

$$\texttt{t (h <*> x)} \equiv \texttt{t h <*> t x}. \tag{9}$$

We define the type of all applicative natural transformations between `f` and `g`, we write, in Haskell,

```
type AppNat f g = ∀a.f a → g a
```

where the laws are implied.
Similarly, for any pair of functors `f` and `g`, we define

```
type Nat f g = ∀a.f a → g a
```

for the type of natural transformations between `f` and `g`.
Note that, by parametricity, polymorphic functions are automatically natural transformations in the categorical sense, i.e, for all

```
t :: Nat f g
h :: a → b
x :: f a,
```

```
t (fmap h x) ≡ fmap h (t x).
```

It is clear that applicative functors, together with applicative natural transformations, form a category, which we denote by $\mathscr{A}$, and similarly, functors and natural transformations form a category $\mathscr{F}$.

Proposition 3. `FreeA` *defines a functor* $\mathscr{F} \to \mathscr{A}$.

Proof. We already showed that `FreeA` sends objects (functors in our case) to applicative functors.
We need to define the action of `FreeA` on morphisms (which are natural transformations in our case):

```
liftT :: (Functor f , Functor g)
      ⇒ Nat f g
      → AppNat (FreeA f) (FreeA g)
```

```
liftT _ (Pure x) = Pure x
liftT k (h :$: x) = k h :$: liftT k x
```

First we verify that `liftT k` is an applicative natural transformation i.e. it satisfies laws 8 and 9. We use equational reasoning for proving law 8:

```
  liftT k (pure x)
≡ ⟨ definition of pure ⟩
  liftT k (Pure x)
≡ ⟨ definition of liftT ⟩
  Pure x
≡ ⟨ definition of pure ⟩
  pure x
```

For law 9 we use induction on the size of the first argument of `(<*>)` as explained in section 8. The base cases:

```
  liftT k (Pure h <*> Pure x)
≡ ⟨ definition of (<*>) ⟩
  liftT k (fmap h (Pure x))
≡ ⟨ definition of fmap ⟩
  liftT k (Pure (h x))
≡ ⟨ definition of liftT ⟩
  Pure (h x)
≡ ⟨ definition of fmap ⟩
  fmap h (Pure x)
≡ ⟨ definition of (<*>) ⟩
  Pure h <*> Pure x
≡ ⟨ definition of liftT ⟩
  liftT k (Pure h) <*> liftT k (Pure x)
```

```
  liftT k (Pure h <*> (i :$: x))
≡ ⟨ definition of (<*>) ⟩
  liftT k (fmap h (i :$: x))
≡ ⟨ definition of fmap ⟩
  liftT k (fmap (h ∘) i :$: x)
≡ ⟨ definition of liftT ⟩
  k (fmap (h ∘) i) :$: liftT k x
≡ ⟨ k is natural ⟩
  fmap (h ∘) (k i) :$: liftT k x
≡ ⟨ definition of fmap ⟩
  fmap h (k i :$: liftT k x)
≡ ⟨ definition of (<*>) ⟩
  Pure h <*> (k i :$: liftT k x)
≡ ⟨ definition of liftT ⟩
  liftT k (Pure h) <*> liftT k (i :$: x)
```

The inductive case:

```
  liftT k ((h :$: x) <*> y)
≡ ⟨ definition of (<*>) ⟩
  liftT k (fmap uncurry h :$: (fmap (,) x <*> y)
≡ ⟨ definition of liftT ⟩
  k (fmap uncurry h) :$: liftT k (fmap (,) x <*> y)
≡ ⟨ inductive hypothesis ⟩
  k (fmap uncurry h) :$:
  (liftT k (fmap (,) x) <*> liftT k y)
≡ ⟨ liftT k is natural ⟩
  k (fmap uncurry h) :$:
  (fmap (,) (liftT k x) <*> liftT k y)
≡ ⟨ k is natural ⟩
  fmap uncurry (k h) :$:
  (fmap (,) (liftT k x) <*> liftT k y)
≡ ⟨ definition of (<*>) ⟩
  (k h :$: liftT k x) <*> liftT k y
≡ ⟨ definition of liftT ⟩
  liftT k (h :$: x) <*> liftT k y
```

Now we need to verify that `liftT` satisfies the functor laws

```
      liftT id ≡ id
 liftT (t ∘ u) ≡ liftT t ∘ liftT u.
```

The proof is a straightforward structural induction. □

We are going to need the following natural transformation (which will be the unit of the adjunction 11):

```
one :: Functor f ⇒ Nat f (FreeA f)
one x = fmap const x :$: Pure ()
```

which embeds any functor `f` into `FreeA f` (we used a specialization of this function for `Option` in section 1.2).

Lemma 4.

```
g :$: x ≡ one g <*> x
```

Proof. Given

```
h :: a → ((), a)
h x = ((), x)
```

it is easy to verify that:

```
(∘ h) ∘ uncurry ∘ const ≡ id,        (10)
```

so

```
  one g <*> x
≡ ⟨ definition of one ⟩
  (fmap const g :$: Pure ()) <*> x
≡ ⟨ definition of (<*>) and functor law for f ⟩
  fmap (uncurry ∘ const) g :$: fmap h x
≡ ⟨ equation 1 and functor law for f ⟩
  fmap ((∘ h) ∘ uncurry ∘ const) g :$: x
≡ ⟨ equation 10 ⟩
  g :$: x
```

□

Proposition 4. *The* `FreeA` *functor is left adjoint to the forgetful functor* $\mathscr{A} \to \mathscr{F}$*. Graphically:*

$$Hom_{\mathscr{F}}(\texttt{FreeA f},\texttt{g}) \underset{\overleftarrow{\texttt{raise}}}{\overset{\overrightarrow{\texttt{lower}}}{\cong}} Hom_{\mathscr{A}}(\texttt{f},\texttt{g}) \tag{11}$$

Proof. Given a functor `f` and an applicative functor `g`, we define a natural bijection between `Nat f g` and `AppNat (FreeA f) g` as such:

```
raise :: (Functor f , Applicative g)
      ⇒ Nat f g
      → AppNat (FreeA f) g
raise _ (Pure x) = pure x
raise k (g :$: x) = k g <*> raise k x
lower :: (Functor f , Applicative g)
      ⇒ AppNat (FreeA f) g
      → Nat f g
lower k = k ∘ one
```

A routine verification shows that `raise` and `lower` are natural in `f` and `g`. The proof that `raise k` satisfies the applicative natural transformation laws 8 and 9 is a straightforward induction having the same structure as the proof that `liftT k` satisfies these laws (proposition 3). To show that `f` and `g` are inverses of each other, we reason by induction and calculate in one direction:

```
  raise (lower t) (Pure x)
≡ ⟨ definition of raise ⟩
  pure x
≡ ⟨ t is an applicative natural transformation ⟩
  t (pure x)
≡ ⟨ definition of pure ⟩
  t (Pure x)
```

```
  raise (lower t) (g :$: x)
≡ ⟨ definition of raise ⟩
  lower t g <*> raise (lower t) x
≡ ⟨ induction hypothesis ⟩
  lower t g <*> t x
≡ ⟨ definition of lower ⟩
  t (one g) <*> t x
≡ ⟨ t is an applicative natural transformation ⟩
  t (one g <*> x)
≡ ⟨ lemma 4 ⟩
  t (g :$: x)
```

The other direction:

```
    lower (raise t) x
  ≡ ⟨ definition of lower ⟩
    raise t (one x)
  ≡ ⟨ definition of one ⟩
    raise t (fmap const x :$: Pure ())
  ≡ ⟨ definition of raise ⟩
    t (fmap const x) <*> pure ()
  ≡ ⟨ t is natural ⟩
    fmap const (t x) <*> pure ()
  ≡ ⟨ fmap h ≡ ((pure h) <*>) in an applicative functor ⟩
    pure const <*> t x <*> pure ()
  ≡ ⟨ t is natural ⟩
    pure ($ ()) <*> (pure const <*> t x)
  ≡ ⟨ applicative law 5 ⟩
    pure (∘) <*> pure ($ ()) <*> pure const <*> t x
  ≡ ⟨ applicative law 6 applied twice ⟩
    pure id <*> t x
  ≡ ⟨ applicative law 4 ⟩
    t x
```

□

7.1 Example: option parsers (continued)

With the help of the adjunction defined above by `raise` and `lower` we are able to define some useful functions. In the case of command-line option parsers, for example, it can be used for computing the global default value of a parser:

```
parserDefault :: FreeA Option a → Maybe a
parserDefault = raise optDefault
```

or for extracting the list of all the options in a parser:

```
allOptions :: FreeA Option a → [String]
allOptions = getConst ∘ raise f
  where
    f opt = Const [optName opt]
```

allOptions works by first defining a function that takes an option and returns a one-element list with the name of the option, and then lifting it to the Const applicative functor.
The raise function can be thought of as a way to define a "semantics" for the whole syntax of the DSL corresponding to FreeA f, given one for just the individual atomic actions, expressed as a natural transformation from the functor f to any applicative functor g.
When defining such a semantics using raise, the resulting function is automatically an applicative natural transformation. In some circumstances, however, it is more convenient to define a function by pattern matching directly on the constructors of FreeA f, like when the target does not have an obvious applicative functor structure that makes the desired function an applicative natural transformation.
For example, we can write a function that runs an applicative option parser over a list of command-line arguments, accepting them in any order:

```
matchOpt :: String → String
         → FreeA Option a
         → Maybe (FreeA Option a)
matchOpt _ _ (Pure _) = Nothing
matchOpt opt value (g :$: x)
  | opt ≡ '-' : '-' : optName g
  = fmap (<$> x) (optReader g value)
  | otherwise
  = fmap (g :$: ) (matchOpt opt value x)
```

The matchOpt function looks for options in the parser which match the given command-line argument, and, if successful, returns a modified parser where the option has been replaced by a pure value.
Clearly, matchOpt opt value is not applicative, since, for instance, equation 8 is not satisfied.

```
runParser :: FreeA Option a
          → [String]
          → Maybe a
runParser p (opt : value : args) =
  case matchOpt opt value p of
    Nothing → Nothing
    Just p′ → runParser p′ args
runParser p [] = parserDefault p
runParser _ _ = Nothing
```

Finally, runParser calls matchOpt with successive pairs of arguments, until no arguments remain, at which point it uses the default values of the remaining options to construct a result.

8 Totality

All the proofs in this paper apply to a total fragment of Haskell, and completely ignore the presence of bottom. The Haskell subset we use can be given a semantics in any locally presentable cartesian closed category.
In fact, if we assume that all the functors used throughout the paper are accessible, all our inductive definitions can be regarded as initial algebras of accessible functors.

For example, to realise `FreeA f`, assume `f` is κ-accessible for some regular cardinal κ. Then define a functor:

$$A : \mathsf{Func}_\kappa(\mathscr{C},\mathscr{C}) \to \mathsf{Func}_\kappa(\mathscr{C},\mathscr{C}),$$

where Func_κ is the category of κ-accessible endofunctors of $\mathscr{C}$, which is itself locally presentable by proposition 5.
The inductive definition of `FreeA f` above can then be regarded as the initial algebra of A, given by:

$$(AG)a = a + \int^{b:\mathscr{C}} F[b,a] \times Gb, \tag{12}$$

where $[-,-]$ denotes the internal hom (exponential) in $\mathscr{C}$. Since F and G are locally presentable, and $\mathscr{C}$ is cocomplete, the coend exists by lemma 8, and AG is κ-accessible by lemma 7, provided κ is large enough.
Furthermore, the functor A itself is accessible by proposition 6, hence it has an initial algebra. Equation 1 is then a trivial consequence of this definition.
As for function definitions, most use primitive recursion, so they can be realised by using the universal property of the initial algebra directly.
One exception is the definition of `(<*>)`:

```
(h :$: x) <*> y = fmap uncurry h :$: ((,) <$> x <*> y)
```

which contains a recursive call where the first argument, namely `(,) <$> x`, is not structurally smaller than the original one (`h :$: x`).
To prove that this function is nevertheless well defined, we introduce a notion of *size* for values of type `FreeA f a`:

```
size :: FreeA f a → ℕ
size (Pure _) = 0
size (_ :$: x) = 1 + size x
```

To conclude that the definition of `(<*>)` can be made sense of in our target category, we just need to show that the size of the argument in the recursive call is smaller than the size of the original argument, which is an immediate consequence of the following lemma.

Lemma 5. *For any function* `f :: a → b` *and* `u :: FreeA f a`,

```
size (fmap f u) ≡ size u
```

Proof. By induction:

```
  size (fmap f (Pure x))
≡ ⟨ definition of fmap ⟩
  size (Pure (f x))
≡ ⟨ definition of size ⟩
  0
≡ ⟨ definition of size ⟩
  size (Pure x)
  size (fmap f (g :$: x))
≡ ⟨ definition of fmap ⟩
  size (fmap (f ∘) g :$: x)
≡ ⟨ definition of size ⟩
  1 + size x
≡ ⟨ definition of size ⟩
  size (g :$: x)
```

□

In most of our proofs using induction we carry out induction on the size of the first argument of `(<*>)` where size is defined by the above `size` function.

9 Semantics

In this section, we establish the results about accessible functors of locally presentable categories that we used in section 8 to justify the inductive definition of `FreeA f`.
We begin with a technical lemma:

Lemma 6. *Suppose we have the following diagram of categories and functors:*

$$\begin{tikzcd}
\mathscr{A} \arrow[d, "i"'] & \\
\mathscr{B} \arrow[r, "F"] \arrow[d, "K"'] & \mathscr{C} \\
\mathscr{D} \arrow[ur, dashed, "L"'] &
\end{tikzcd}$$

where $\mathscr{B}$, and $\mathscr{C}$ are locally presentable, F is accessible, and i is the inclusion of a dense small full subcategory of compact objects of $\mathscr{B}$. Then the pointwise left Kan extension L of F along K exists and is equal to the left Kan extension of Fi along Ki.
If, furthermore, $\mathscr{D}$ is locally presentable and K is accessible, then L is accessible.

Proof. The pointwise left Kan extension L can be obtained as a colimit:

$$Ld = \operatorname*{colim}_{\substack{b:\mathscr{B} \\ g:Kb\to d}} Fb, \tag{13}$$

Where the indices range over the comma category $(K \downarrow d)$. To show that L exists, it is therefore enough to prove that the colimit 13 can be realised as the small colimit:

$$\operatorname*{colim}_{\substack{a:\mathscr{A} \\ f:Ka\to d}} Fa.$$

For any $b : \mathscr{B}$, and $g : Kb \to d$, we can express b as a canonical filtered colimit of compact objects:

$$b \cong \operatorname*{colim}_{\substack{a:\mathscr{A} \\ h:a\to b}} a.$$

Since F preserves filtered colimits, we then get a morphism:

$$Fb \to \operatorname*{colim}_{\substack{a:\mathscr{A}\\ h:a\to b}} Fa \to \operatorname*{colim}_{\substack{a:\mathscr{A}\\ f:Ka\to d}} Fa.$$

This gives a cocone for the colimit 13, and a straightforward verification shows that it is universal.
As for the second statement, we can assume that Ka is compact for all $a : \mathscr{A}$. Then, by the first part:

$$Ld = \int^{a:\mathscr{A}} Fa \cdot \mathscr{D}(Ka, d).$$

Now, a filtered colimit in d commutes with $\mathscr{D}(Ka, -)$ because Ka is compact, it commutes with $Fa \cdot -$ because copowers are left adjoints, and it commutes with coends because they are both colimits.
Therefore, L is accessible. □

From now on, let $\mathscr{B}$ and $\mathscr{C}$ be categories with finite products, and $F, G : \mathscr{B} \to \mathscr{C}$ be functors.

Definition 2. *The* Day convolution *of F and G, denoted $F * G$, is the pointwise left Kan extension of the diagonal functor in the following diagram:*

$$\begin{tikzcd}
\mathscr{B} \times \mathscr{B} \arrow[r, "F \times G"] \arrow[d, "\times"'] \arrow[dr] & \mathscr{C} \times \mathscr{C} \arrow[d, "\times"] \\
\mathscr{B} \arrow[r, dashed, "F * G"'] & \mathscr{C}
\end{tikzcd}$$

Note that the Day convolution of two functors might not exist, but it certainly does if $\mathscr{B}$ is small and $\mathscr{C}$ is cocomplete.

Lemma 7. *Suppose that $\mathscr{B}$ and $\mathscr{C}$ are locally presentable and F and G are accessible. Then the Day convolution of F and G exists and is accessible.*

Proof. Immediate consequence of lemma 6. □

Lemma 8. *Suppose that $\mathscr{B}$ is cartesian closed. Then the Day convolution of F and G can be obtained as the coend:*

$$(F * G)b = \int^{y:\mathscr{B}} F[y, b] \times Gy$$

Proof. By coend calculus:

$$\begin{aligned}
&(F * G)b \\
&= \int^{xy:\mathscr{B}} Fx \times Gy \cdot \mathscr{C}(x \times y, b) \\
&= \int^{y:\mathscr{B}} \left(\int^{x:\mathscr{B}} Fx \cdot \mathscr{C}(x \times y, b) \right) \times Gy \\
&= \int^{y:\mathscr{B}} \left(\int^{x:\mathscr{B}} Fx \cdot \mathscr{C}(x, [y, b]) \right) \times Gy \\
&= \int^{y:\mathscr{B}} F[y, b] \times Gy
\end{aligned}$$

□

Proposition 5. *Let κ be a regular cardinal, and $\mathscr{B}$ and $\mathscr{C}$ be locally κ-presentable. Then the category $\mathsf{Func}_\kappa(\mathscr{B},\mathscr{C})$ of κ-accessible functors is locally κ-presentable.*

Proof. Let $\mathscr{A}$ be a dense small full subcategory of $\mathscr{B}$. The obvious functor $\mathsf{Func}_\kappa(\mathscr{B},\mathscr{C}) \to \mathsf{Func}(\mathscr{A},\mathscr{C})$ is an equivalence of categories (its inverse is given by left Kan extensions along the inclusion $\mathscr{A} \to \mathscr{B}$), and $\mathsf{Func}(\mathscr{A},\mathscr{C})$ is locally κ-presentable (see for example [2], corollary 1.54). $\square$

Proposition 6. *Let κ be a regular cardinal such that $\mathscr{B}$ and $\mathscr{C}$ are locally κ-presentable, and the Day convolution of any two κ-accessible functors is κ-accessible (which exists by lemma 7).*
Then the Day convolution operator

$$\mathsf{Func}_\kappa(\mathscr{B},\mathscr{C}) \times \mathsf{Func}_\kappa(\mathscr{B},\mathscr{C}) \to \mathsf{Func}_\kappa(\mathscr{B},\mathscr{C})$$

is itself a κ-accessible functor.

Proof. It is enough to show that $*$ preserves filtered colimits pointwise in its two variables separately. But this is clear, since filtered colimits commute with finite products, copowers and coends. $\square$

We can recast equation 12 in terms of Day convolution as follows:

$$AG = \mathsf{Id} + F * G. \tag{14}$$

Equation 14 makes precise the intuition that free applicative functors are in some sense lists (i.e. free monoids). In fact, the functor A is exactly the one appearing in the usual recursive definition of lists, only in this case the construction is happening in the monoidal category of accessible endofunctors equipped with Day convolution.
We also mention the following purely categorical construction of free applicative (i.e. lax monoidal) functors, which is not essential for the rest of the paper, but is quite an easy consequence of the machinery developed in this section.
The idea is to perform the "list" construction in one step, instead of iterating individual Day convolutions using recursion. Namely, for any category $\mathscr{C}$, let $\mathscr{C}^*$ be the *free monoidal category* generated by $\mathscr{C}$. The objects (resp. morphisms) of $\mathscr{C}^*$ are lists of objects (resp. morphisms) of $\mathscr{C}$. Clearly, $\mathscr{C}^*$ is locally presentable if $\mathscr{C}$ is.
If $\mathscr{C}$ has finite products, there is a functor

$$\varepsilon : \mathscr{C}^* \to \mathscr{C}$$

which maps a list to its corresponding product. Note that ε is accessible. Furthermore, the assigment $\mathscr{C} \to \mathscr{C}^*$ extends to a 2-functor on Cat which preserves accessibility of functors.
Now, the free applicative G on a functor $F : \mathscr{C} \to \mathscr{C}$ is simply defined to be the Kan extension of $\varepsilon \circ F^*$ along ε:

$$\begin{tikzcd}
\mathscr{C}^* \arrow[r, "F^*"] \arrow[d, "\varepsilon"'] \arrow[dr] & \mathscr{C}^* \arrow[d, "\varepsilon"] \\
\mathscr{C} \arrow[r, dashed, "G"'] & \mathscr{C}
\end{tikzcd}$$

The functor G is accessible by lemma 7, and it is not hard to see that it is lax monoidal (see for example [7], proposition 4). We omit the proof that G is a free object, which can be obtained by diagram chasing using the universal property of Kan extensions.

10 Related work

The idea of free applicative functors is not entirely new. There have been a number of different definitions of free applicative functor over a given Haskell functor, but none of them includes a proof of the applicative laws.

The first author of this paper published a specific instance of applicative functors[2] similar to our example shown in section 1.2. The example has later been expanded into a fully-featured Haskell library for command line option parsing.[3]

Tom Ellis proposes a definition very similar to ours,[4] but uses a separate inductive type for the case corresponding to our `(:$:)` constructor. He then observes that law 6 probably holds because of the existential quantification, but does not provide a proof. We solve this problem by deriving the necessary equation 1 as a "free theorem".

Gergő Érdi gives another similar definition[5], but his version presents some redundancies, and thus fails to obey the applicative laws. For example, `Pure id <*> x` can easily be distinguished from `x` using a function like our `count` above, defined by pattern matching on the constructors.

However, this is remedied by only exposing a limited interface which includes the equivalent of our `raise` function, but *not* the `Pure` and `Free` constructors. It is probably impossible to observe a violation of the laws using the reduced interface, but that also means that definitions by pattern matching, like the one for our `matchOpt` in section 7.1, are prohibited.

The `free` package on hackage[6] contains a definition essentially identical to our `FreeAL`, differing only in the order of arguments.

Another approach, which differs significantly from the one presented in the paper, underlies the definition contained in the `free-functors` package on hackage,[7] and uses a Church-like encoding (and the `ConstraintKinds` GHC extension) to generalize the construction of a free `Applicative` to any superclass of `Functor`.

The idea is to use the fact that, if a functor T has a left adjoint F, then the monad $T \circ F$ is the codensity monad of T (i.e. the right Kan extension of T along itself). By taking T to be the forgetful functor $\mathscr{A} \to \mathscr{F}$, one can obtain a formula for F using the expression of a right Kan extension as an end.

One problem with this approach is that the applicative laws, which make up the definition of the category $\mathscr{A}$, are left implicit in the universal quantification used to represent the end.

In fact, specializing the code in `Data.Functor.HFree` to the `Applicative` constraint, we get:

```
data FreeA′ f a = FreeA′ {
  runFreeA :: ∀g. Applicative g
           ⇒ (∀x. f x → g x) → g a}
instance Functor f ⇒ Functor (FreeA′ f) where
  fmap h (FreeA′ t) = FreeA′ (fmap h ∘ t)
instance Functor f ⇒ Applicative (FreeA′ f) where
  pure x = FreeA′ (λ_ → pure x)
```

[2] http://paolocapriotti.com/blog/2012/04/27/applicative-option-parser
[3] http://hackage.haskell.org/package/optparse-applicative
[4] http://web.jaguarpaw.co.uk/~tom/blog/posts/2012-09-09-towards-free-applicatives.html
[5] http://gergo.erdi.hu/blog/2012-12-01-static_analysis_with_applicatives/
[6] http://hackage.haskell.org/package/free
[7] http://hackage.haskell.org/package/free-functors

```
FreeA′ t1 <*> FreeA′ t2 =
  FreeA′ (λu → t1 u <*> t2 u)
```

Now, for law 4 to hold, for example, we need to prove that the term `λu → pure id <*> t u` is equal to `t`. This is strictly speaking false, as those terms can be distinguished by taking any functor with an `Applicative` instance that does not satisfy law 4, and as `t` a constant function returning a counterexample for it.

Intuitively, however, the laws should hold provided we never make use of invalid `Applicative` instances. To make this intuition precise, one would probably need to extend the language with quantification over equations, and prove a parametricity result for this extension.

Another problem of the Church encoding is that, like Érdi's solution above, it presents a more limited interface, and thus it is harder to use. In fact, the destructor `runFreeA` is essentially equivalent to our `raise` function, which can only be used to define *applicative* natural transformation. Again, a function like `matchOpt`, which is not applicative, could not be defined over `FreeA′` in a direct way.

11 Discussion and further work

We have presented a practical definition of free applicative functor over any Haskell functor, proved its properties, and showed some of its applications. As the examples in this paper show, free applicative functors solve certain problems very effectively, but their applicability is somewhat limited.

For example, applicative parsers usually need an `Alternative` instance as well, and the free applicative construction does not provide that. One possible direction for future work is trying to address this issue by modifying the construction to yield a free `Alternative` functor, instead.

Unfortunately, there is no satisfactory set of laws for alternative functors: if we simply define an alternative functor as a monoid object in $\mathscr{A}$, then many commonly used instances become invalid, like the one for `Maybe`. Using rig categories and their lax functors to formalise alternative functors seems to be a workable strategy, and we are currently exploring it.

Another direction is formalizing the proofs in this paper in a proof assistant, by embedding the total subset of Haskell under consideration into a type theory with dependent types.

Our attempts to replicate the proofs in Agda have failed, so far, because of subtle issues in the interplay between parametricity and the encoding of existentials with dependent sums.

In particular, equation 1 is inconsistent with a representation of the existential as a Σ type in the definition of `FreeA`. For example, terms like `const () :$: Pure 3` and `id :$: Pure ()` are equal by equation 1, but can obviously be distinguished using large elimination.

This is not too surprising, as we repeatedly made use of size restrictions in sections 8 and 9, and those will definitely need to be somehow replicated in a predicative type theory like the one implemented by Agda.

A reasonable compromise is to develop the construction only for *containers* [1], for which one can prove that the free applicative on the functor $S \triangleright P$ is given, using the notation of section 9, by $S^* \triangleright (\varepsilon \circ P^*)$, where S is regarded as a discrete category.

Another possible further development of the results in this paper is trying to generalize the construction of a free applicative functor to functors of any monoidal category. In section 9 we focused on categories with finite products, but it is clear that monoidal categories are the most natural setting, as evidenced by the appearance of the corresponding 2-comonad on Cat.

Furthermore, an applicative functor is defined in [6] as a lax monoidal functor *with a strength*, but we completely ignore strengths in this paper. This could be remedied by working in the more general setting of $\mathcal{V}$-categories and $\mathcal{V}$-functors, for some monoidal category $\mathcal{V}$.

12 Acknowledgements

We would like to thank Jennifer Hackett, Thorsten Altenkirch, Venanzio Capretta, Graham Hutton, Edsko de Vries and Christian Sattler, for helpful suggestions and insightful discussions on the topics presented in this paper.

References

[1] Michael Abott, Thorsten Altenkirch & Neil Ghani (2005): *Containers - Constructing Strictly Positive Types*. Theoretical Computer Science 342, pp. 3–27, doi:10.1016/j.tcs.2005.06.002. Applied Semantics: Selected Topics.

[2] J. Adámek & J. Rosický (1994): *Locally Presentable and Accessible Categories*. Cambridge University Press, doi:10.1017/CBO9780511600579.

[3] Brian J. Day (1970): *Construction of Biclosed Categories*. Ph.D. thesis, University of New South Wales.

[4] G. M. Kelly (1980): *A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on*. Bulletin of the Australian Mathematical Society 22, pp. 1–83, doi:10.1017/S0004972700006353.

[5] Simon Marlow (2010): *Haskell 2010 Language Report*.

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

[7] Ross Paterson (2012): *Constructing Applicative Functors*. In: Mathematics of Program Construction, Lecture Notes in Computer Science 7342, Springer-Verlag, pp. 300–323, doi:10.1007/978-3-642-31113-0_15.

[8] John C. Reynolds (1983): *Types, Abstraction and Parametric Polymorphism*. In: IFIP Congress, pp. 513–523.

[9] S. Doaitse Swierstra & Luc Duponcheel (1996): *Deterministic, Error-Correcting Combinator Parsers*. In: Advanced Functional Programming, Lecture Notes in Computer Science 1129, Springer-Verlag, pp. 184–207, doi:10.1007/3-540-61628-4_7.

[10] Wouter Swierstra (2008): *Data types à la carte*. Journal of Functional Programming 18(4), pp. 423–436, doi:10.1017/S0956796808006758.

[11] Philip Wadler (1989): *Theorems for free!* In: Functional Programming Languages and Computer Architecture, ACM Press, pp. 347–359, doi:10.1145/99370.99404.