Modelling Call-Time Choice as Effect using Scoped Free Monads

Niels Bunkenburg

Master's Thesis

Programming Languages and Compiler Construction
Department of Computer Science
Kiel University

Advised by
Priv.-Doz. Dr. Frank Huch
M. Sc. Sandra Dylus

March 4, 2019

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Aus fremden Quellen direkt oder indirekt übernommene Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ort, Datum

Unterschrift

Abstract

Contents

1 Introduction

2 Preliminaries

2.1 Coq

- Introduce the necessary Coq concepts to understand the paper

2.2 Haskell

- Introduce the necessary Haskell concepts to understand the paper

2.2.1 Monad and MonadPlus

2.3 Curry

- Introduce the necessary Curry concepts to understand the paper

2.3.1 Non-strictness

2.3.2 Sharing

2.3.3 Non-determinism

2.4 Modelling Curry Programs using Monadic Code Transformation

- Why is the naive MonadPlus approach not sufficient to model Curry semantic?
- Motivate usage of monadic data types
- Introduce explicit sharing

Modelling Curry programs in a language like Haskell requires a transformation of non-deterministic code into a semantically equivalent, deterministic program. First, we have a look at the direct representation of non-determinism used in the KiCS2 implementation as described by Braßel et al. [2011].

2.4.1 KiCS2 Approach

Non-determinism in Curry is not limited to flat non-determinism but can occur within components of data structures and anywhere in a computation. This means that expressing non-determinism via Haskell's list monad is not sufficient to model Curry's non-determinism. Instead, existing data types receive additional constructors that represent failure and the choice between two values. For example, the extended list data type looks as follows. TODO: Example

```
data List a = Nil | Cons a (List a) | Choice (List a) (List a) | Fail
```

Since this transformation adds new constructors, all functions need to cover these cases, too. The new rules return `Fail` if the function's argument is a failed computation and distribute function calls to both branches if the argument is a choice.

One issue with this approach is that call-time choice is not implemented yet. If a choice is duplicated during evaluation, this information cannot be recovered later. Therefore, each `Choice` constructor has an additional `ID` argument that identifies the same choices. Since each choice needs a fresh ID, functions use an additional `IDSupply` argument when choices are created.

The evaluation of a non-deterministic value is implemented by transforming the value into a search tree which can be traversed with different search strategies. In the process, each choice ID's decision is stored and then repeated if the same ID is encountered again.

While this approach is useful when the host language supports laziness and sharing, another approach is necessary to model these effects when they are not built into the language.

2.4.2 Modelling Laziness and Sharing

Fischer et al. [2009] introduce a monadic representation of non-determinism that supports sharing and non-strict evaluation. Out of simplicity, the implementation idea is presented in Haskell, similar to the approach of the original authors, using the example of permutation sort. The algorithm consists of three components. Firstly, a function `insert` that inserts an element non-deterministically at every possible position within a list.

```
insert :: MonadPlus m => a -> [a] -> m [a]
insert x xs = return (x:xs)
      `mplus` case xs of
                []     -> mzero
                (y:ys) -> do zs <- insert x ys
                             return (y:zs)
```

The second part is the function `perm` that inserts the head of a given list into the permutations of the list's tail.

```
perm :: MonadPlus m => [a] -> m [a]
perm [] = return []
```

```
perm (x:xs) = do ys <- perm xs
                 zs <- insert x ys
                 return zs
```

Finally, the function `sort` generates permutations and then tests whether they are sorted.

```
sort :: MonadPlus m => [Int] -> m [Int]
sort xs = do ys <- perm xs
             guard (isSorted ys)
             return ys
```

The function `isSorted` compares each element in a list to the next one to determine whether the list is sorted. When we test this implementation, we can see that the runtime increases significantly when adding even a few elements.

```
λ> sort [9, 8..1] :: [[Int]]
[[1,2,3,4,5,6,7,8,9]]
(0.69 secs)
λ> sort [10, 9..1] :: [[Int]]
[[1,2,3,4,5,6,7,8,9,10]]
(6.67 secs)
λ> sort [11, 10..1] :: [[Int]]
[[1,2,3,4,5,6,7,8,9,10,11]]
(77.54 secs)
```

The reason for the factorial runtime is that the implementations is needlessly strict. A list of length n has $n!$ permutations, all of which are generated when running `sort`. This matches our observation above, since adding a tenth element increases the runtime by a factor of 10 and an eleventh element multiplies the runtime of the ten-element list by eleven.

If we consider the implementation of `isSorted`, we can see that, as soon as the comparison of two elements yields `False`, the function returns `False` and does not evaluate the remainder of the list.

```
isSorted :: [Int] -> Bool
isSorted (x:y:zs) = (x <= y) && isSorted (y:zs)
isSorted _        = True
```

However, since we use bind to pass permutations from `perm` to `isSorted`, each permutation is fully evaluated before it is determined whether the permutation is sorted. This leads to the complete evaluation of every permutation, which results in an inefficient program.

Similarly, when we consider the Curry example `head (1 : head [] : [])`, the strictness of our `MonadPlus` approach shows again. The corresponding Haskell expression is as follows.

```
hd [] >>= \x -> hd (1 : x : [])
```

Here `hd :: MonadPlus a => [a] -> m a` is the lifted `head` function. Evaluating the expression in Haskell yields `mzero`, that is, no result, while Curry returns `1`. The reason is the definition of the bind operator. For example, the monad instance for lists defines bind as `xs >>= f = concatMap f xs`. In the expression above, this means that the pattern matching within `concatMap` evaluates `hd []` to `mzero` and thus returns `mzero`.

The strictness observed in both examples is the motivation for an alternative approach. The problem with the above implementations is that non-deterministic arguments of constructors need to be evaluated completely before the computation can continue. Therefore, we would like to be able to use unevaluated, non-deterministic computations as arguments of constructors.

As mentioned before, we can implement this idea by adapting all data types so that they may contain non-deterministic components.

```
data List m a = Nil | Cons (m a) (m (List m a))
```

The list data type now has an additional argument `m` of type `* -> *` that represents a non-determinism monad. Instead of fixed constructors like `Choice`, the monad `m` determines the structure and evaluation strategy of the non-determinism effect. Two smart constructors `cons` and `nil` make handling the new list type more convenient.

```
nil :: Monad m => m (List m a)
nil = return Nil

cons :: Monad m => m a -> m (List m a) -> m (List m a)
cons x y = return (Cons x y)
```

Adapting the permutation sort functions to the lifted data type requires us to replace `[]` with `List m` However, this is not sufficient because the list itself can be the result of a non-deterministic computation. Therefore, an additional `m` is wrapped around every occurrence of `List`.

```
insert' :: MonadPlus m => m a -> m (List m a) -> m (List m a)
insert' mx mxs = cons mx mxs
  `mplus` mxs >>= \xs -> case xs of
                           Nil         -> mzero
                           Cons my mys -> cons my (insert' mx mys)

perm' :: MonadPlus m => m (List m a) -> m (List m a)
perm' ml = ml >>= \l ->
  case l of
    Nil -> nil
    Cons mx mxs -> insert' mx (perm' mxs)
```

Whenever pattern matching occurred in the original definition, we now use bind to extract a `List` value. Since this only evaluates flat non-determinism and not non-determinism that occurs in the components, non-strictness is upheld as much as possible.

All functions now take arguments of the same type they return. Thus, the definition of `sort` does not need bind in order to pass permutations to `isSorted`.

```
sort' :: MonadPlus m => m (List m Int) -> m (List m Int)
sort' xs = let ys = perm' xs in
  isSorted' ys >>= \sorted -> guard sorted >> ys
```

We are now able to take advantage of `isSorted`'s non-strict definition. The implementation generates permutations only if there is a chance that the permutation is sorted, that is, only recursive calls of `perm` that are demanded by `isSorted` are executed.

We reconsider the Curry example `head (1 : head [] : [])`. Since the `List` data type now takes monad values as arguments, we can write the example using the smart constructors and a lifted `head` function as follows.

```
λ> hd' (cons (return 1) (cons (hd' nil) nil))
1
```

Because we do not need to use bind to get the result of `hd' nil`, the expression is not evaluated due to non-strictness and the result is equal to Curry's output.

Data types with non-deterministic components solve the problem of non-strictness because each component can be evaluated individually, instead of forcing the evaluation of the whole term. Unfortunately, this leads to a problem. When unevaluated components are shared via Haskell's built-in sharing, computations, rather than results, are being shared. This means that the results can be different each time the computation is evaluated, which contradicts the intuition of sharing.

The solution to this problem is an explicit sharing combinator `share :: m a -> m (m a)` that allows sharing the results of a computation in a non-strict way. Here, `m` is a `MonadPlus` instance, similar to the monad used in the definition of the data type, that supports sharing. Thus, `share` takes a computation and then returns a computation that returns the result, that is, the shared value. The reason for this nesting of monad layers is that, in short, the `share` combinator performs some actions that can be immediately executed by bind (the outer monad layer), while the inner monad layer should only be evaluated when needed. This is explained in more detail later. With the explicit sharing operator we can adapt `perm'` to share the generated permutations in order to achieve non-strictness in combination with sharing.

```
sort' :: MonadPlus m => m (List m Int) -> m (List m Int)
sort' xs = do ys <- share (perm' xs)
              sorted <- isSorted'
              guard sorted
              ys
```

The `share` operator must satisfy certain laws, which we discuss in section 4.4. The implementation of `share` is subject of the next chapter.

3 Call-Time Choice modelled in Haskell

Based on the ideas presented in the last chapter, we now want to model call-time choice, that is, non-strictness, sharing and non-determinism, in Haskell. We still use `MonadPlus` to parameterize our programs. However, instead of, for example, using the list instance to make non-determinism visible, we define an effect functor that can express many different effects, including non-determinism and sharing. This approach, as introduced by Wu et al. [2014], will also be the base of the Coq implementation shown in chapter 4.

TODO: Definition effect

For the implementation of call-time choice, we want to be able to express different effects within our programs. However, not every program contains effects. There are also pure programs that have no side-effects besides the computation of a value. A data type that represents such programs could look as follows.

```
data Void a = Return a
```

Here, `Void` means the absence of effects.

If we consider programs that contain effects, also called impure, like, non-determinism, a data type that represents such values could look like the following.

```
data ND a = Return a
          | Fail
          | Choice (ND a) (ND a)
```

This data type also has a constructor to model pure values but in addition, there are constructors that represent failed computations and the non-deterministic choice between two values. We could go on and list data types that model many more effects but the question is: Is it possible to create a data type that, if appropriately instantiated, behaves like the original effect functor? This would allow us to represent programs with many different effects using one compact data type.

Answering this question requires abstracting the concrete form of effect functors into a general program data type. As we saw in the examples above, we need a way to represent pure values in a program. Therefore, the first constructor of our new program data type should be `Return a` for the type `a`, that is, the result type of the program. To model effects like non-determinism, the program is parameterized over effect functors of type `* -> *` that represent, for example, `Fail` and `Choice`. We call this argument `sig` because the signature of a program tells us which effects can occur. So far, programs are defined as `data Prog sig a = Return a`. In order do make use of the `sig` component, we need to add a constructor for impure operations. The `ND` data type shows us that effect functors can be defined recursively. Thus, the constructor for impure programs should be recursive, too, to be able to represent this structure.

```
data Prog sig a = Return a | Op (sig (Prog sig a))
```

With this definition of `Prog`, we are able to represent the original functors by instantiating `sig` appropriately. For `Void`, we already have the `Return` constructor. Therefore, the data type we can use with `Prog` does not need a constructor anymore, that is, `data Void' a`.

```
VoidProg a = Return a
           | Op (Void' (VoidProg a)) -- Void' has no constructors!
```

The type `Prog Void'` now resembles the original type `Void` since the `Op` constructor would require a value of type `Void'`, which we cannot construct. [1] Only `Return` can be used to define values, similar to the original data type.

Similar to `Void'`, we can define a data type `Choice` that represents `Choice` in combination with `Prog`.

```
data ND p = Fail | Choice p p
```

Again, we can omit the `Return` constructor because it is already part of the `Prog` data type. For the same reason, the type variable `a` has been replaced with the variable `p`, since `ND` does not have values as arguments but rather programs that return values.

```
data NDProg a = Return a
              | Op (Choice (NDProg a))
```

Since `Op` applies `sig` recursively, this yields the following type, which is equivalent to the original data type.

```
data NDProg a = Return a
              | OpFail
              | OpChoice (NDProg a) (NDProg a)
```

We have found a way to model effect functors as instances of the data type `Prog`, which essentially models a tree with leafs, represented by the `Return` constructor, and branches that have the form defined by `sig`.

TODO: Tree structure visualization?

3.1 Free Monads

- What are free monads?
- Why do we use free monads?

The data type `Prog` is better known as the free monad. We saw in the previous chapter that `Free` can be used to model other data types. In addition, `Free` is a monad that can turn any functor into a monad.

[1] It is possible to use `undefined` to create an impure value of type `Prog Void' a`. Since this is not possible in Coq, we do not consider this in the Haskell implementation.

We consider, for example, the type `Free One` where `data One a = One`. Here `a` is a phantom type that we need because `Free` expects a functor. The monad instance for `Free` is as follows.

```
data Free f a = Pure a | Impure (f (Free f a))

instance (Functor f) => Monad (Free f) where
  return = Pure
  Pure x >>= g = g x
  Impure fx >>= g = Impure (fmap (>>= g) fx)
```

Since `One` has only a single, non-recursive constructor `One`, the only possible impure value is `Impure One`, whereas the usual `Return` constructor remains. If bind encounters the value `One`, the function `g` is distributed deeper into the term structure using `fmap`. Since `fmap One = One`, it becomes apparent that the monad constructed by `Free One` is the `Maybe` monad.

TODO: Visualization of fmap tree

Since we want to model different effects in our program, the free monad makes writing programs easier by allowing monadic definitions without defining a separate monad instance for each effect.

TODO: Find more reasons for using free monads

3.2 Modelling Effects

TODO: Stacking monads does not necessarily yield monads again

- Explanation of the Prog/sig infrastructure
- ND and state effect implementation

TODO: Split ND into choice and fail instead of one

In the previous sections the free monad and its ability to represent effect functors was discussed. The goal of this section is to explore the infrastructure that allows us to combine multiple effects, write effectful programs and compute the result of such programs.

3.2.1 Combining Effects

Firstly, we would like to combine multiple effects. For this purpose, we use the technique introduced by Swierstra [2008] to define a data type that combines the effect functors `sig1` and `sig2`. The infix notation simplifies combining multiple effects via nested applications of `:+:`.

```
data (sig1 :+: sig2) a = Inl (sig1 a) | Inr (sig2 a)
```

For example, the type `ND :+: One` is a functor that we can use with `Prog` to define programs that contain non-determinism and partiality as follows.

```
progNDOne :: Prog (ND :+: One) Int
progNDOne = Op (Inl (Choice (Op (Inr One)) (Return 42)))
```

In the example `progNDOne` we define a program that represents the non-deterministic choice between a program whose value is absent and a program that returns `42`. The complexity of nesting constructors of `Prog` and `:+:` correctly increases quickly for bigger terms. Therefore, we define a type class that allows us to define such expressions more conveniently. The class is parameterized over two functors, one of which is a subtype – regarding `:+:` – of the other.

```
class (Functor sub, Functor sup) => sub :<: sup where
  inj :: sub a -> sup a
```

We need a few instances of the class `:<:` to make it useful. The simplest case is `sig :<: sig` where want to inject a value of type `sig a` into the same type. Since we do not need to modify the value in any way, `id` is used to define `inj`.

```
instance Functor sig => sig :<: sig where
  inj = id
```

The next instance covers the case `sig1 :<: (sig1 :+: sig2)`. Since we already know that `sig1` is part of the sum type, we only need to apply the correct constructor of `:+:`, that is, `Inl` because `sig1` is the left argument.

```
instance (Functor sig1, Functor sig2) => sig1 :<: (sig1 :+: sig2) where
  inj = Inl
```

The last instance assumes that we can inject `sig` into `sig2` and describes how we can inject `sig` into `sig1 :+: sig2`. In this case, we can use `inj` to receive a value of type `sig2 a`. All that remains is a situation similar to the previous instance, where we only need to use the matching constructor to complete the injection.

```
instance (Functor sig1, sig :<: sig2) => sig :<: (sig1 :+: sig2) where
  inj = Inr . inj
```

These instances allow us to write a polymorphic definition of the function `inject` which injects constructors depending on the given type of the program.

```
inject :: sig1 :<: sig2 => sig1 (Prog sig2 a) -> Prog sig2 a
inject = Op . inj
```

`inject` can then be used as demonstrated in the following example.

```
λ> inject One :: Prog (One :+: ND) a
Op (Inl One)
λ> inject One :: Prog (ND :+: One) a
Op (Inr One)
```

The implementation of the function `inject` assumes that we can inject `sig1` into `sig2`. This is because `sig2` is the signature of the returned program and `sig1` is the type of the effect constructor that we want to inject. This restriction is justified because, for example, non-deterministic syntax should only appear in a program where `ND` is part of the signature. With this part of the infrastructure in place, we can redefine the example `progNDOne` without using `Inl` and `Inr` explicitly.

```
progNDOne' :: Prog (ND :+: One) Int
progNDOne' = inject (Choice (inject One) (Return 42))
```

Deriving the appropriate instance of `:<:` when using `inject` is, however, not always unambiguous. The last two instances overlap in situations where `sig = sig1`. For example, the example `inject One :: Prog (One :+: One) a` yields different values with respect to the chosen constructor of `:<:`, depending on the instance.

```
λ> nothing
Op (Inl One) -- second instance
λ> nothing
Op (Inr One) -- third instance
```

This is because the type constraint of `inject`, in this case `One :<: (One :+: One)`, matches both the second and third instance. Haskell does not accept overlapping instances by default, which is why we prioritize one instance via pragmas. In practice, the different term structure due to `Inl` and `Inr` does not influence the evaluation as long as we do not explicitly match for the constructors. This is ensured by an additional function `prj` of the type class `:<:`, which is discussed in the next section.

3.2.2 Simplified Pattern Matching

While the function `inject` allows us to write programs in a more convenient way, we also need to consider how we can evaluate programs. The same issue of nested applications of `Op`, `Inl` and `Inr` applies when we want to distinguish different effects via pattern matching. Thus, we add a second function `prj` to the type class `:<:`.

```
class (Functor sub, Functor sup) => sub <: sup where
  inj :: sub a -> sup a
  prj :: sup a -> Maybe (sub a)
```

The function `prj` is a partial inverse to `inj`. This means that we can project values of a type `sup a` into a subtype `sub a`. For this reason, the return type of the function is a `Maybe` type. Similar to `inj`, we have to define instances for the same cases as before.

- For `sig :<: sig`, we can define `prj` as `Just` because we know that every element of the supertype is also an element the subtype.

- `sig1 :<: (sig1 :+: sig2)` means that we can return `Just x` for `Inl x`. However, for `Inr` we need to return `Nothing` because we cannot, in general, project from `sig2` to `sig1`.

- In the last case `sig :<: sig2 => sig :<: (sig1 :+: sig2)` we know that we can project from `sig2` to `sig`. Thus, in case of `Inr x`, where `x` has the type `sig2`, we can apply `prj` to construct a value of appropriate type. The other case `prj (Inl _)` is handled by returning `Nothing`.

With the definition of `prj` and the instances of `:<:`, we can now define the function `project` which we can use to make pattern matching more convenient.

```
project :: (sub :<: sup) => Prog sup a -> Maybe (sub (Prog sup a))
project (Op s) = prj s
project _      = Nothing
```

Due to the recursive definition of the `Prog` data type, constructors like `Choice` have `Prog` arguments themselves. Thus, `sub` is applied to `Prog sup a` in the return type of the projection. We can only project effectful values because generally it is not clear which functor we should choose for `sub` when projecting a `Return` value.

Finally, we can now inject and project effectful values. Since `project` is a partial inverse of `inject`, the equation `project (inject x) = Just x` holds for values `x` of appropriate type, excluding failing computations. This is demonstrated in the following example.

TODO: Does it hold?

```
λ> type T = Maybe (ND (Prog (ND :+: One) Int))
λ> project (inject (Choice (Return 42) (Return 43))) :: T
Just (Choice (Return 42) (Return 43))
```

Now that we can use `project` as an abstraction of the concrete term structure regarding `:<:`, we can write a first function that evaluates a non-deterministic, partial program.

```
evalNDOne :: Prog (ND :+: One) a -> [a]
evalNDOne (Return x) = [x]
evalNDOne p = case project p of
               Just (Choice p1 p2) -> evalNDOne p1 ++ evalNDOne p2
               Just Fail           -> []
               Nothing             -> case project p of
                                        Just One -> []
                                        Nothing  -> []
```

When `evalNDOne` encounters a value `Return x`, `x` is returned as a singleton list. For effectful programs, we can use `project` to distinguish between the constructors of one effect at a time. The case patterns hold the necessary type information for `project`. When the projection returns `Nothing`, another effect can be matched in a nested case expression. Since we never need to explicitly match for `Inl` or `Inr`, overlapping patterns in the instances of `:<:` do not affect the evaluation of programs in our model.

Although we have already eliminated `Inl`, `Inr` and `Op` from functions that create or evaluate programs, there can be done even more to simplify programming with effects. Two language extensions, `PatternSynonyms` and `ViewPatterns`, allow us to write definitions like the following.

```
pattern PChoice p q <- (project -> Just (Choice p q))
```

View patterns – the right-hand side of the `<-`, make pattern-matching for certain cases more compact. A view pattern consists of a function on the left-hand side of `->`, that is applied to the value that the pattern is matched against, and a pattern on the right-hand side. The result of the function call is matched against this pattern and the variables inside the pattern can be used in the definition. The function `evalNDOne` can be defined using view patterns in the following way.

```
evalNDOne' :: Prog (ND :+: One) a -> [a]
evalNDOne' (Return x) = [x]
evalNDOne' (project -> Just (Choice p1 p2)) = evalNDOne' p1 ++ evalNDOne' p2
evalNDOne' (project -> Just Fail          ) = []
evalNDOne' (project -> Just One           ) = []
```

We cannot use (`project -> Nothing`) without type annotations as a pattern because this would result in overlapping instances. However, no effects other than those specified in the signature can occur within the program. Therefore, the `Nothing` pattern is not necessary.

The second component of the pattern definition above is the option to define a synonym for more complex patterns. In this case, we name the view patterns similar to the original constructors of the effects. While this is necessary for every effect constructor, it allows us to rewrite the definition in the following way.

```
evalNDOne'' :: Prog (ND :+: One) a -> [a]
evalNDOne'' (Return    x) = [x]
evalNDOne'' (PChoice p q) = evalNDOne'' p ++ evalNDOne'' q
evalNDOne'' (PFail      ) = []
evalNDOne'' (POne       ) = []
```

Writing programs that evaluate effectful programs is now almost as convenient as simple pattern matching. Finally, a useful definition for working with programs that have the signature `f :+: g`, where want to match for `f` but not `g`, is as follows.

```
pattern Other s = Op (Inr s)
```

Since `:+:` is right-associative in nested applications, we can match for the left argument effect and conveniently match all remaining effects with `Other`.

3.2.3 Effect Handlers

For each effect in a program's signature, a handler is required. Handling an effect means transforming a program that contains a certain effect into a program where the effect's syntax does not occur anymore. However, the syntax is not just removed, but the effect's semantics is applied. The semantics of an effect is therefore given by its handler. In the following we discuss handlers for the effects non-determinism and state.

Void Effect We begin with the data type for absence of effects, `Void`. Due to its definition without constructors, there is no `Void` syntax that needs to be handled. The only constructor for programs with the signature `Void` is `Return`, which we can handle by returning the argument. Thus, the handler for `Void` removes the program layer and is usually applied last, when all other effects have been handled.

```
run :: Prog Void a -> a
run (Return x) = x
```

Non-determinism Effect We already defined a data type for non-deterministic programs in chapter 3. The `Choice` constructor did not contain any IDs, which we need for the implementation of call-time choice. Thus, the revised data type is as follows.

```
data ND p = Fail | Choice (Maybe ID) p p
```

Not every non-deterministic choice in a program needs an ID, since IDs slow down the evaluation of choices considerably. Thus, IDs are optional and only assigned when necessary, that is, when choices are shared.

In the last section, we already defined a function `evalNDOne` that handles the simple `ND` type without IDs by returning a list of results, where, for each choice, the result lists are concatenated. For choices with IDs, however, this is not sufficient. We begin by transforming the program into a program that returns a tree data type which mirrors the non-determinism structure.

TODO: Keep tree structure?

```
runND :: (Functor sig) => Prog (ND :+: sig) a -> Prog sig (Tree.Tree a)
runND (Return a) = return (Tree.Leaf a)
runND Fail       = return Tree.Failed
runND (Choice m p q ) = do
  pt <- runND p
  qt <- runND q
  return (Tree.Choice m pt qt)
runND (Other op) = Op (fmap runND op)
```

Next, we need to memorize the decisions that were made while traversing the choice tree. For this reason, we define a data type `Decision` that indicates whether the left or right branch of a choice has been picked before for a particular choice ID. A `Memo` is maps IDs to decisions.

```
data Decision = L | R
type Memo = Map.Map ID Decision
```

The depth-first traversal of the choice tree is implemented in the function `dfs`. The returned list of results is created similar to the approach in `evalNDOne`, except for the case where a choice has a non-empty ID. The ID could have appeared in a choice that is closer to the root node of the tree and thus, the choice could have already been decided. Therefore, we need to look up the ID in the `Memo`. If the choice has not been made yet, that is, `Nothing` is returned, the `Memo` is updated with `L` for the left branch and `R` for the right branch. The recursive calls then descend into the corresponding branch and will make the same decision for this ID if it occurs again. If, on the other hand, a decision is returned by the `lookup` function, the branch of the recursive call is chosen according to the decision.

```
dfs :: Memo -> Tree a -> [a]
```

```
dfs mem Failed = []
dfs mem (Leaf x) = [x]
dfs mem (Choice Nothing t1 t2) = dfs mem t1 ++ dfs mem t2
dfs mem (Choice (Just n) t1 t2) =
    case Map.lookup n mem of
      Nothing -> dfs (Map.insert n L mem) t1
              ++ dfs (Map.insert n R mem) t2
      Just L -> dfs mem t1
      Just R -> dfs mem t2
```

The function `dfs` is called with an empty map and yields the list of results that the choice tree represents.

TODO: Examples

State Effect Stateful computations are an important part of the sharing effect that is presented in section 3.3. We begin by defining the syntax of the state effect. Usually, stateful computations can read the current state with `get` and set a new state with `put`. Thus, the data type needs those two constructors, too. We add an additional type variable that abstracts the type of values that the state can hold. The variable `p` represents the program type as before.

```
data State s p = Get' -- ?
               | Put' -- ?
```

Both constructors have an effect on a program, that is, a scope in which the effects are visible. Thus, both constructors need a program argument `p`. For `Put'`, we can simply add the arguments `s` for the new state and `p` for the program in which the new state is set. The constructor `Get'`, however, contains the program in a different form, namely a function `s -> p`. The reason for this is that, if we were using a simple `p` argument, the handler would have to somehow replace all `get`-occurrences of the state with appropriate values. This would require evaluating the whole program, which would defeat the purpose of preserving non-strictness. Hence, the program is added to `get` in the form of a functional expression where the function argument replaces the occurrences of the state that are being read in the program. The data type for the state effect now looks as follows.

```
data State s p = Get' (s -> p)
               | Put' s p
```

The smart constructors for stateful programs are defined by instantiating the program and function arguments appropriately. For `get`, this means that we need to supply a function of type `s -> Prog sig s` since `p` is `Prog sig s` in this context. Conveniently, the `return` function matches this type and thus, is the initial argument of `Get'`. For `put`, the new state and a program that returns `()` are supplied to `Put'` because `put` does not return any information.

```
get :: (State s :<: sig) => Prog sig s
get = inject (Get' return)
```

```
put :: (State s :<: sig) => s -> Prog sig ()
put s = inject (Put' s (return ()))
```

The choice of initial function arguments might not seem intuitive at first because it is not clear how the remaining program finds its way into the argument of, for example, `Get'`. Therefore, we consider an example of the state effect and how the free monad is used to write programs.

```
p :: Prog (State Int :+: Void) Int
p = do put 42
       i <- get
       return (i * 2)
```

The program sets a state `42`, gets the value of the current state and then returns double of that. The normal form of `p` can be computed by evaluating the occurrences of bind. We recall the monad instance for the free monad: bind uses `fmap` to distribute a function deeper into a term. Thus, we first define a `Functor` instance.

```
instance (Functor sig) => Monad (Prog sig) where
  return x = Return x
  Return x >>= f = f x
  Op op >>= f = Op (fmap (>>= f) op)

instance Functor (State s) where
  fmap f (Get' g)   = Get' (f . g)
  fmap f (Put' s p) = Put' s (f p)
```

In the case of `Get'`, we need to apply `g` to a state in order to obtain a program that we can apply `f` to. Thus, we pass the result from `f` to `g` via function composition. For `Put'`, the state `s` remains unmodified and the function `f` is applied to the program argument `p` of the constructor.

Now we can transform the program `p` into normal form as follows.

```
put 42 >>= \_ -> get >>= \i -> return (i * 2)
= inject $ fmap (>>= \_ -> get >>= \i -> return (i * 2)) (Put' 42 (return ()))
= inject $ Put' 42 (get >>= \i -> return (i * 2))
= inject $ Put' 42 (inject $ fmap (>>= \i -> return (i * 2)) (Get' return))
= inject $ Put' 42 (inject $ (Get' (\i -> return (i * 2))))
```

`Op` as well as `Inl` and `Inr` constructors are replaced by `inject` in this example. The expression is transformed by applying the definitions of bind and `fmap`. In the last step, we simplify the expression by applying the left identity monad law, that is, `(>>= f) . return = f`.

We can now see that the remaining program after `get`, that is, the `return` call, has been moved into the argument function of `Get'`. The function expects a state and replaces the variables, that were bound to the return value of `get` in the original program, with the state.

Now that we have seen the definition of stateful program syntax and how the state flows through the program via functions, we can define the handler for the state effect.

Naturally, the handler needs to keep track of the current state, which is the first argument of the function. Then, the function expects a program that contains state syntax. Finally, the return type is a program that returns a pair of the current state and a return value.

```
runState :: Functor sig => s -> Prog (State s :+: sig) a -> Prog sig (s, a)
runState s (Return a) = return (s, a)
runState s (Get     k) = runState s (k s)
runState s (Put s' k) = runState s' k
runState s (Other op) = Op (fmap (runState s) op)
```

For pure values, the current state and the value inside the `Return` constructor is returned. When a `Get` is encountered, we apply the function argument, which expects a state, to the current state and do a recursive call with the resulting program. `Put` is handled by a recursive call where the old state is replaced by the new state while the program stays the same. Finally, other syntax is handled by using `fmap` to distribute the handler deeper into the term structure, similar to the other handlers we have seen.

The example program `p` can now be handled by first calling the handler `runState` to handle the state effect, followed by `run` to extract the result from the program structure.

```
λ> run . runState 1 $ p
(42,84)
```

As expected, the first component represents the current state, which was set by `put` to `42`, while the second component is the result that was returned after multiplying the current state by two.

3.2.4 Handling Order

When multiple effects are part of the signature, the question arises whether running handlers in a different order has an effect on the result. As an example, we define a handler that does not remove syntax but actually adds state syntax to a non-deterministic program. The function `results` keeps the structure of a program intact but adds state syntax that increments the current state by one for each result.

```
results :: (ND <: sig, State Int <: sig) => Prog sig a -> Prog sig a
results (Return x)       = get >>= put . (+ 1) >> return x
results Fail             = fail
results (Choice m p q ) = choiceID m (results p) (results q)
results (Op op)          = Op (fmap results op)
```

Now we define a program `tree` that builds the complete, binary choice tree of height `x`. For each call of `tree`, a choice is made where the current height is either incremented or decremented by one.

```
tree :: (ND <: sig) => Int -> Prog sig Int
tree 0 = return 0
tree x = tree (x - 1) >>= \i ->
  choice (return $ i + 1) (return $ i - 1)
```

Each time choice is called, two new branches are created. Thus, we expect `tree x` to have 2^x results. To see the program in action, we define two handlers. The difference between `treeGlobal` and `treeLocal` is the order of the handlers. In both cases `results` is run first, but whereas `treeGlobal` runs the non-determinism handler before the state handler, the opposite is true for `treeLocal`.

```
treeGlobal :: (Int, Tree.Tree Int)
treeGlobal = run . runState 0 . runND . results $ tree 2

treeLocal :: Tree.Tree (Int, Int)
treeLocal = run . runND . runState 0 . results $ tree 2
```

The types of the definitions already indicate a difference. While `treeGlobal` returns a state paired with a tree of results, `treeLocal` returns a tree of state and result pairs. In the following, the result of evaluating each handler chain is presented as a visualization of the resulting choice tree.

```
λ> putStrLn . pretty $ treeGlobal
(4,?
|
├── ?
|   |
|   ├── 2
|   |
|   └── 0
|
└── ?
    |
    ├── 0
    |
    └── -2)
```

```
λ> putStrLn . pretty $ treeLocal
?
|
├── ?
|   |
|   ├── (1,2)
|   |
|   └── (1,0)
|
└── ?
    |
    ├── (1,0)
    |
    └── (1,-2)
```

As the name suggests, `treeGlobal`, that is, handling non-determinism first and state second, evaluates the program with a global state, where each non-deterministic branch shares the same state. Contrary to that, `treeLocal` creates an individual state for every non-deterministic branch by handling state syntax first. While the results are not influenced by the order of handlers in this case, this is not generally the case.

3.3 Implementing Scoped Effects

- How can we implement simple sharing as an effect?
- What about deep/nested sharing?
- Examples (exRecList, ...)

Although Haskell offers sharing as part of the language, we have seen in subsection 2.4.2 that the built-in sharing mechanism does not always work as intended when combined with lifted data types. Thus, we need to model sharing as an effect using the tools that were presented in the previous section. There is, however, a difference between sharing and the other effects we have seen so far. Sharing is not an independent effect since it affects non-deterministic choices. This means that, depending on the presence of sharing, some choice branches may not be explored. Therefore, sharing is a scoped effect, that is, only a delimited part of the program is affected by the effect.

Wu et al. [2014] present two ways to define scoped effects. Firstly, syntax for explicitly marking the begin and end of a scope can be defined. This leads to a more complicated handler because the begin and end tags can be mismatched in the program and one needs to keep track of the current scope environment. The second approach uses higher-order syntax, that is, the signature of a program is not just a functor but a function that takes a functor as an argument. This approach makes it possible to have the scoped program as an argument of the syntax constructor. In the following, an overview of a – initially promising but ultimately incorrect – hybrid approach and both options mentioned before is given.

3.3.1 Hybrid Implementation

The idea of the hybrid implementation is a combination of the explicit scoping infrastructure and direct program arguments in the syntax definition that the higher-order implementation uses. In theory, this has the benefit of simple handlers and scoping via program arguments instead of explicit tags. Therefore, it seemed worthwhile to explore this approach instead of following one of the options mentioned in the introduction of the section.

Beginning with the definition of the sharing syntax data type, we follow the idea of the higher-order approach and define a single constructor `Share'` with a program argument that represents the shared program. Although `p` is supposed to be only the shared program, the monadic bind structure moves the program that follows the `Share'` constructor into the argument `p` The same happened in ?? for the program argument of the state effect constructor `put`.

```
data Share p = Share' p

share :: (Share :<: sig) => Prog sig a -> Prog sig (Prog sig a)
share p = return $ inject (Share' p)
```

The return type of `share` is not just a program but a program that returns a program. The reason for this is explained later in section 3.4. For the first implementation of `share`, this outer program layer is empty and thus created by `return`.

In order to create an example that showcases the usage of `share` and the monadic structure, we need a few definitions. First, we define a non-deterministic coin that returns either `0` or `1` and a lifted addition function for programs that return integer results. Since `(+)` is a strict function in Haskell and Curry, we can mirror this behavior by binding both program arguments and then adding the results.

```
coin :: (ND :<: sig) => Prog sig Int
coin = choice (return 0) (return 1)

addM :: (Functor sig) => Prog sig Int -> Prog sig Int -> Prog sig Int
addM p q = p >>= \i -> q >>= \j -> return (i + j)
```

With these functions defined, we can now use the `share` operator to add a shared coin to an unshared coin, twice, as shown in the following example. This corresponds to the Curry code `let x = coin in (x + coin) + (x + coin)`.

```
exAddSharedCoinTwice :: Prog (Share :+: ND :+: Void) Int
exAddSharedCoinTwice = share coin >>= \fx -> addM (addM fx coin)
                                                  (addM fx coin)
```

```
< ?1
 ├─ ?2
 │  ├─ < ?1
 │  │   ├─ ?2
 │  │   │  ├─ 0 > >
 │  │   │  └─ 1 > >
 .  .   └─ ?2
 .  .      ├─ 1 > >
 .  .      └─ 2 > >
```

Hybrid implementation

```
< ?1
 ├─ > ?
 │    ├─ < ?1
 │    │    ├─ > ?
 │    │    │    ├─ 0
 │    │    │    └─ 1
 .    .    └─ > ?
 .    .         ├─ 1
 .    .         └─ 2
```

Explicit scoping tags

The left-hand side tree is generated using the data type `Share` with a single constructor, while the right-hand side visualizes a data type with two constructors that explicitly delimit the scope. Subscript numbers represent the ID of a choice. Although this information is added by a sharing handler we have not defined yet, choice IDs are important in order to understand the consequences of the data type definitions for the sharing effect.

Choice IDs are assigned inside a sharing scope. When a sharing scope is duplicated due to the monadic structure, the choices inside get the same IDs. Finally, when the choice tree is evaluated, these choices are linked. The right-hand side tree shows us that explicit scoping tags allow ending a scope in a program. For example, the scope around the root choice ends first and then the next scope is opened. The visualization of the hybrid term shows that all opened sharing scopes are only closed at the end of each branch. This difference in term structure means that the handler for the hybrid approach never stops assigning IDs to choices because it cannot distinguish the shared program that was initially passed as an argument and the following program that was moved into the argument by the monadic structure.

The hybrid implementation correctly assign the ID `1` for the choice that immediately follows the beginning of the scope. This is the shared choice that is defined in `coin`. The next choice within the branch originates from the unshared coin and ideally should not receive an ID. Indeed, the implementation with explicit `begin` and `end` tags closes the sharing scope after the first choice and thus, the choice does not receive an ID. The hybrid implementation, however, cannot stop assigning IDs to choices and thus assigns `2` to the choice.

In the hybrid implementation, when a new scope is opened, the current scope is overwritten. For this example, this means that the next choice is labeled with `1` again, since each scope is associated with an initial state that is copied, too, when the sharing scope is duplicated. Because there is only one sharing scope in the original program, all occurring scopes are duplicates that were created due to non-determinism. It is critical that copied sharing scopes behave identical because this ensures that the choices inside the scopes are named the same way, resulting in correct call-time choice behavior. In the example, however, this leads to a fatal flaw. Until now, assigning the ID `2` to the unshared choice below the root choice was unnecessary but not incorrect. As a consequence of the second sharing scope behaving identical to the first one, the second unshared choice also receives the ID `2`. Since we now have two equal IDs within a branch, this means that the second choice with the ID `2` is linked to the decision of the first choice with the ID `ID`, that is, the first unshared `coin` is linked to the second one.

This was not intended in the original program and proves that the hybrid approach is unsuitable for modelling scoped effects and, consequently, sharing. Interestingly, this approach promisingly passed all example tests and algorithms in both Haskell and Coq. The flaw was only found while doing the finishing touches on the ID generation algorithm. This shows that the hybrid approach is not incorrect in its entirety but merely requires some refinement, as shown in the next subsection.

3.3.2 Higher-Order Scope Syntax

The higher-order approach described by Wu et al. [2014] is based on a modified program data type to represent scoped syntax. So far, the type variable `sig` has been a functor that is applied to the program type again. In the higher-order data type, however, `sig` is applied to a program functor and a type, which makes it of type `(* -> *) -> * -> *`.

```
data Prog   sig a = Return a | Op (sig (Prog sig a))
data ProgHO sig a = Return a | Op (sig (Prog sig) a)
```

Due to the functor argument of `sig`, it is now called a higher-order functor. Based on the new program type and higher-order functors, the existing infrastructure for combining signatures, injecting values and pattern matching can be adapted. This is not discussed here since we are mostly interested in the the definition of effect data types. For example, the higher-order version of the sharing effect is defined as follows.

```
data HShare m a = forall x. Share' (m x) (x -> m a)
```

Due to the new type of `sig` in the definition of programs, effect data types have an additional argument now, too. The single argument `p` has been replaced by a functor argument `m` and a type `a`. Applying `m` to `a` corresponds to the argument `p` we have seen in the previous effect types. One advantage of splitting `p` is that it is now possible to apply `m` to different types, whereas we were limited to `p` before. Wu et al. [2014] demonstrate that this can be useful, for example, when defining exceptions with `throw` and `catch` syntax. Syntax for `catch` usually consists of a program where exceptions may occur, a handler for said exceptions and the remaining program. This structure is very similar to the sharing effect since we would also like to pass the shared program as an argument to the sharing syntax. However, this was not possible with functor-based program type, as we have seen in the previous subsection. With higher-order programs, however, we can represent the shared program as an argument of type `m x` where `m` represents `Prog sig` and `x` the return value of the program. The remaining program is a continuation function `x -> m a`that takes the result of the shared program and substitutes the results of all matching calls of `share`, similar to how the current state is propagated in the program for the state effect.

The purpose of forall x lies in adding an independent type variable using the language extension `ExistentialQuantification`. In this case, independence means that the variable does not occur on the left-hand side of the definition and thus can be different for two values of the same type. For example, the following data type has a regular type variable and one introduced by `forall`.

```
data Test a = forall x. Test x a

instance Functor Test where
  fmap f (Test x a) = Test x (f a)
```

With this definition, `[Test 42 True, Test () False]` is a valid expression of type `Test Bool`. When we define a functor instance for `Test`, the argument `x` remains unmodified while `f` is applied to `a`. Although in a different form, this applies to the sharing data type as well. The call of `fmap`, or rather the higher-order equivalent `emap`, in the definition of bind is responsible for building the program structure and thus, appends the remaining program to the shared program in the case of the definition we used for the hybrid implementation. Since `emap` transforms a value of type `Share m a` into a `Share m b`, there is no way to leave one program argument (the shared program) unmodified while applying a function to the other. For this reason, the additional, independent type variable `x` is necessary in the definition of the sharing effect data type.

One disadvantage of the higher-order approach is the more complicated infrastructure and effect handlers. In short, `Other` cases are harder to handle because the simple `fmap`-approach does not work anymore. Additionally, due to the function argument of `Share'`, the visualization of sharing scopes and programs becomes difficult. Therefore, we will pursue the explicit scoping syntax approach for the remainder of the Haskell chapter.

3.3.3 Explicit Scope Syntax

The previous subsections have demonstrated that program arguments do not correctly model scopes unless we use higher-order infrastructure. Thus, an alternative approach is needed. A well known syntactical structure for delimiting scopes are explicit scope tags in the form of `begin` and `end` or brackets. Following this idea, we split the sharing syntax into two parts. One constructor marks the beginning of the scope, while the other marks the ending of the scope.

```
data Share p = BShare' p | EShare' p
```

Both constructor have programs arguments. `BShare'`s argument program contains the scoped program block followed by an `Eshare'` with the remaining program as an argument. Similar to the state effect, our smart constructors use `return ()` as an initial program that is replaced by the actual program when the bind structure is evaluated.

```
begin :: (Share :<: sig) => Prog sig ()
begin = inject (BShare' (return ()))

end :: (Share :<: sig) => Prog sig ()
end = inject (EShare' (return ()))
```

For example, the following expression shows a scope that includes the `Choice'` constructor but not the `Return` values.

```
inject $ BShare' (inject $ Choice' Nothing
                                    (inject $ EShare' (Return 0))
                                    (inject $ EShare' (Return 1)))
```

Now that we can delimit the scope of the sharing effect, it is time to define the actual sharing operator.

```
share :: (Share :<: sig) => Prog sig a -> Prog sig (Prog sig a)
share p = return $ do begin ; x <- p ; end ; return x
```

`share` wraps `begin` and `end` tags around a call of bind that executes the program `p`. Then, the result is returned. One problem of this approach is that sharing tags can be mismatched. For this reason, sharing syntax should only be accessible by means of the smart constructor `share`. Nevertheless, mismatched scoping tags are part of the syntax definition and need to be handled.

Now that we have defined the syntax of the sharing effect with explicit scope constructors, we need to consider how the handler should work. From the structure of the syntax follows that the handler needs to extract the scoped program between the `begin` and `end` tags and then modify the choices that occur inside the scope. Following this idea, we divide the sharing handler into two parts. The first part is `bshare`, a function that waits for a `begin` tag and then hands over its program argument to `eshare`, which handles the scope and finally returns the program that follows the scope. Since this program is now outside of the scope, `bshare` waits for the next `begin` tag without modifying any choices.

```
bshare :: (ND <: sig) => Prog (Share + sig) a -> Prog sig a
bshare (Return a) = return a
bshare (BShare p) = eshare p >>= bshare
bshare (EShare p) = error "mismatched Eshare"
bshare (Other op) = Op (fmap bshare op)
```

The case of mismatched scoping tags, that is, an `Eshare` occurring before a `BShare` has opened a scope, can be handled in Haskell with a run-time error. In Coq, however, this is not possible. We could wrap the return type of the function in `Maybe` to represent mismatched tags, but this makes proofs more cumbersome due to the added case distinction. A solution to this problem is discussed in the next chapter about modelling call-time choice in Coq.

The second part of the handler handles the scoped program and thus should modify choices in such a way that the program behaves as expected regarding call-time choice.

```
eshare :: (ND <: sig)
       => Prog (Share + sig) a -> Prog sig (Prog (Share + sig) a)
eshare (Return a)     = return (Return a)
eshare (BShare p)     = eshare p
eshare (EShare p)     = return p
eshare Fail           = fail
eshare (Choice _ p q) = choiceID {- ID? -} (eshare p) (eshare q)
eshare (Other op)     = Op (fmap eshare op)
```

Pure values are simply returned. When a `begin` tag is found, this means that there is a scope within in scope, that is, nested scopes. In this case, `eshare` keeps modifying choices because neither the original scope nor the new one has not been closed yet. Contrary to that, closing tags result in switching back to `bshare` for the remaining program. Finally, when a choice is encountered, an ID needs to be created for `choiceID`, a function which creates a choice with an explicitly passed ID. However, this is a problem.

The ID that the choice came with is always `Nothing` because choices are created without IDs. It comes to mind that `eshare` could have a state that is incremented for each encountered choice. Unfortunately, this would entail that each choice is assigned a different ID, that is, two choices could never have the same ID. This defeats the purpose of choice IDs because it makes sharing impossible.

Consequently, the main finding from the first attempt to define the sharing handler is that we need to add an identifier to sharing scopes. This allows linking scopes that were duplicated due to non-determinism in the program and can be used to create choice IDs. Since the problem of linking scopes is more relevant to the implementation of the sharing effect than scoped effects in general, it is discussed in the next section.

3.4 Implementation of Sharing as Effect

In this section, the simple implementation of sharing from the previous section is refined into an implementation that models call-time choice correctly.

3.4.1 Sharing IDs

To begin with, we consider the following example that shows why we need to link sharing scopes.

```
exAddSharedCoin :: Prog (Share :+: ND) Int
exAddSharedCoin = share coin >>= \fx -> addM fx fx
```

The `coin` in the addition is shared and thus, the expected result is `0` and `2`. When represented as a tree, the example looks like the following.

```
< ?
```

```
├── > < ?
│       ├── > 0
│       └── > 1
└── > < ?
        ├── > 1
        └── > 2
```

In order to evaluate the example correctly, all choices need to have the same ID. Since all scopes are copies of the same call to `share`, the sharing handler needs to behave equally for all scopes and the choices within. However, this information is lost when the bind structure in the term duplicates the sharing scopes. Hence, the `begin` and `end` tags of the scope receive an ID. Although it would be sufficient to mark only the `begin` tags, it makes checking for mismatched tags easier to give `end` an ID, too.

```
data Share p = BShare' Int p | EShare' Int p
```

With this new data type, how do we define the smart constructor `share`? There are two options: `share` either receives an ID as a parameter or the ID is generated inside the function. The former is much simpler to implement but would entail that the user needs to assign a unique ID to each call of `share`. Since it is good practice to hide such implementation-specific details from the user, the second approach of generating an ID within `share` is the better option.

In order to generate an ID for a sharing scope, we need a state that the ID is derived from. Again, we have two options. The state could be implemented on the level of the modelling language or the modelled language. The former would mean that all programs would need to be defined within the state monad, which is conceptually similar to the approach of user-defined IDs that are put into the program from the outside.

The latter approach uses the state effect on the `Prog` level, which was discussed in ??. This means that `share` itself becomes a complex program instead of a simple smart constructor. In this case, the ID is generated within the program.

Generally, using the `Prog` state effect is preferable because it does not require adapting the whole infrastructure to the state monad and it ties in elegantly with the theme of modelling effects.

```
share :: (Share :<: sig, State Int :<: sig)
      => Prog sig a -> Prog sig (Prog sig a)
share p = return $ do
  i <- get
  put (i + 1)
  begin i
  x <- p
  end i
  return x
```

The signature of the program now needs to support an integer state in order to support sharing syntax. We still use `return` to create an empty, outer program layer. The inner program now contains state syntax that retrieves and increments the current state. The value from the state is then used as the ID of the sharing scope.

The consequence of the added state code is visualized by means of the initial example `addSharedCoin`.

```
do fx <- share coin
   addM fx fx
```

Inlining the definition of `share` yields the following program.

```
do fx <- return $ do
     i <- get
     put (i + 1)
     begin i
     x <- coin
     end i
     return x
   addM fx fx -- state code is duplicated!
```

Due to the left identity law for bind, `fx <- return $ ...` acts like a `let` binding where `fx` is bound to the program that follows `return`. This results in the state code being duplicated in the addition. Unfortunately, this is not the desired behavior, as the following visualization shows.

```
<0 ?
   ├── 0> <1 ?
   │          ├── 1> 0
   │          └── 1> 1
   └── 0> <1 ?
              ├── 1> 1
              └── 1> 2
```

When the state is initialized with `0`, the first scope receives the ID `0` and increments the state to `1` when the state code within the first occurrence of `fx` is executed. Then, the second `fx` is evaluated and the same happens again. Thus, the ID of the following scope is `1` for both branches[2]. Since the idea of the added state is to link scopes together, so that duplicated scopes receive the same ID, this approach has failed. Luckily, just a small modification is needed to fix the problem. The problem of the current `share` implementation is that one part of the program – the state code – needs to be executed immediately, while the other part – the shared program – should only be evaluated if needed. In the current implementation of `share`, there is an empty, outer program layer that is evaluated by bind when using `share`. The reason for the nested program structure now becomes clear: The outer program layer contains the state code that is executed once when bind evaluates `share`.

[2]In this example, the state handler runs before the non-determinism handler and thus, choice branches have a local state.

```
do fx <- do -- state code is executed
     i <- get
     put (i + 1)
     return $ do
       begin i
       x <- coin
       end i
       return x
   addM fx fx
```

Consequently, all occurrences of the state, that is, the scope IDs, are defined before the shared program is evaluated. Thus, it does not matter if or where in the program the result of `share` is evaluated. This is also reflected in the visualization of the example `addSharedCoin`.

```
<0 ?
   ├── 0> <0 ?
   │         ├── 0> 0
   │         └── 0> 1
   └── 0> <0 ?
             ├── 0> 1
             └── 0> 2
```

3.4.2 Sharing Infrastructure

3.4.3 Nested Sharing

With the current definition of the `share` operator, simple sharing examples are modelled correctly. However, there are more complex scenarios that have not been considered yet. For example, calls of `share` within a shared expression, that is, nested sharing, leads to incorrect behavior. We consider the following example of adding the shared result of the addition of a shared coin.

```
exAddSharedCoinNested :: Prog (Share :+: ND) Int
exAddSharedCoinNested = share (share coini >>= \fx -> addM fx fx) >>=
                          \fy -> addM fy fy
```

The problematic part is generating the ID for the inner call of `share`. Whereas the outer sharing scope correctly receives the ID `0` for both occurrences within the term structure, the ID of the inner scope differs.

```
<0 <1 ?
      ├── 1> <1 ?
      │         ├── 1> 0> <0 <2 ?
      .         │                ├── 2> <2 ?
      .         .                │         ├── 2> 0> 0
```

```
     .         .               .         └── 2> 0> 1
```

The scope with ID `0` originate from the outer call of `share`, while the inner scopes correspond to the nested call. Both scopes with the ID `0` should behave identically, including the nested scopes. However, in the current implementation this is not the case. When `fy` is evaluated for the first time, the inner call to `share` receives the ID `1`, since state was incremented by the first call. The following scope with ID `0` is not affected by this because its ID was assigned together with the first scope. The second nested scope is not linked to the first one, however, because the state code of both scopes is executed separately. Thus, the increment operation from running the first nested `share` affects the second one and the ID `2` is assigned, although it should have been `1`.

The problem is this example is therefore that the nested `share` calls are duplicated but the state is not. To solve this problem, we can add `put` to the program before `x <- p`, so that nested calls of `share` in `p` behave identical if the scope program is duplicated.

```haskell
share :: (Share :<: sig, State Int :<: sig)
      => Prog sig a -> Prog sig (Prog sig a)
share p = do
  i <- get
  put i + 1
  return $ do
    begin i
    put {- new state? -}
    x <- p
    end i
    return x
```

For an example like `exAddSharedCoinNested`, the new state can be as simple as `i + 1`. With the added `put` syntax, the state within the duplicated scope is no longer different to the state in the original scope. Hence, the nested scope receives the correct ID.

```
<0 <1 ?
      ├── 1> <1 ?
      │         ├── 1> 0> <0 <1 ?
      .         │                ├── 1> <1 ?
      .         .                │         ├── 1> 0> 0
      .         .                .         └── 1> 0> 1
```

This is not a universal solution, however, since ID clashes can occur in some situations. When nested sharing is followed by another `share` call, as in the following example, the IDs inside the nested `share` and the IDs after the nested share can clash.

```haskell
exAddSharedCoin4 :: Prog (Share :+: ND) Int
exAddSharedCoin4 =
  share (share coin >>= \fx -> addM fx fx) >>=
  \fy -> share coin >>= \fz -> addM fy fz
```

The tree shows the scope 1 wrapped around the duplicated, nested `share` scopes with ID 2. After that, another scope with ID 2 follows, although this scope belongs to the shared coin `fz`.

```
<1 <2 ?
      ├── 2> <2 ?
      │         ├── 2> 1> <2 ?
      .         │            ├── 2> 0
      .         .            └── 2> 1
```

This clash occurred because nested sharing and repeated sharing have the same namespace when `put (i + 1)` is used to set the scope state. In order to make the namespaces unique, one option is to have `put (i * 2)` in the outer program layer and `put (i * 2 + 1)` for the inner program layer. In the adapted syntax tree we can now see that the nested calls have the ID `2 * 1 + 1 = 3`, while the repeated call received the ID `2 * 1 = 2`. Most importantly, the IDs of the nested scopes and the last scope are different now.

```
<1 <3 ?
      ├── 3> <3 ?
      │         ├── 3> 1> <2 ?
      .         │            ├── 2> 0
      .         .            └── 2> 1
```

The `*2`/`*2+1` approach is used, for example, in the KiCS2 compiler. It can lead to large numbers very quickly, however, and is not suitable for Coq due to its Peano representation of numbers. A more elegant solution can be implemented using a pair of integers as state. This way, one component is incremented in the outer program layer and the other component in the inner layer.

3.4.4 Deep Sharing

The implementation of `share` from the previous subsection supports nested sharing and top-level non-determinism. Modelling Curry's call-time choice also includes non-determinism that occurs in components of data types. Therefore, when a value of a data type with non-deterministic components is shared, the individual components should be shared, too. Similarly to subsection 2.4.2, data types need to be lifted so that effectful components can be modelled properly. Since `Prog sig` is a monad if `sig` is a functor, the same monadic transformation works here, too. We reconsider the following lifted list data type.

```
data List m a = Nil | Cons (m a) (m (List m a))

cons :: Monad m => n a -> n (List n a) -> m (List n a)
cons x xs = return (Cons x xs)

nil :: Monad m => m (List n a)
nil = return Nil
```

Handling Effectful Components In order to make the existing infrastructure compatible with effectful components of data structures, we need to think about the way handlers work. Since values of lifted data types are considered pure values, although the components might be effectful, effect handlers do not modify such values, that is, the contained effects are not handled. Instead of differentiating primitive and complex pure values inside all handlers, we choose a different approach. For example, we consider the following transformation of a non-deterministic list in Curry syntax.

```
  [0 ? 1, 0 ? 1]
= [0, 0 ? 1] ? [1, 0 ? 1]
= [0, 0] ? [0, 1] ? [1, 0] ? [1, 1]
```

Beginning with a list that contains non-deterministic elements, we can move choices from the components to the root of the expression. In the end, only lists without effectful arguments remain. The same concept can be transferred to our model. Before running any handlers, effects need to be moved outside of the components, which is called normal form. This concept is formalized in the following type class.

TODO: normal form strictness?

```
class Normalform m a b where
    nf :: m a -> m b
```

The parameters a and b are used to adapt the return type. For example, the function nf can normalize an argument of type `Prog sig (List (Prog sig) a)` into a value of type `Prog sig (List Identity a)`. This means that the effects that were contained in the `Prog sig` argument of `List` are moved into the outer program layer, while the inner program layer is replaced with the identity monad.

The instance of `nf` for lists implements this idea. Firstly, the list is retrieved from the monadic value using bind, followed by pattern matching. The empty list cannot be further normalized. A non-empty list is normalized by recursive calls of `nf` for the element and the remaining list. The results need to be retrieved again because the result of `nf` is a monadic value of the monad `n`, while `m` is expected. Thus, the `return` statements in the last line move the results into the new monad.

```
instance (Normalform n a b, Monad m, Monad n) =>
    Normalform n (List n a) (List m b) where
    nf mxs = mxs >>= \xs ->
               case xs of
                 Nil -> nil
                 Cons mz mzs -> nf mz  >>= \z ->
                                nf mzs >>= \zs ->
                                   cons (return z) (return zs)
```

With `nf` as a normalization layer between effect handlers and data types with effectful components, we can now evaluate expressions like `cons coin (cons coin nil)`, as the following choice tree demonstrates.

```
?
├── ?
```

```
│    ├── [0,0]
│    └── [0,1]
└── ?
     ├── [1,0]
     └── [1,1]
```

This tree represents the transformation in Curry shown above. For all data types that occur in a program, a lifted version with a `Normalform` instance needs to be defined. Primitive types require instances, too, but can be simply defined as `nf = id` because primitive types cannot contain effectful values.

Sharing Complex Data Types At the moment, complex data types like lists can only be shared wholly. This means that expressions like `let xs = [0?1] in xs ++ xs` correctly yield `[0,0]` and `[1,1]`, but `let xs = [0?1] in head xs + head xs` is evaluated to all four possible results instead of only `0` and `2`, as shown below.

```
<(1,1) (1,1)> ?
               ├── <(1,1) (1,1)> ?
               │                  ├── 0
               │                  └── 1
               └── <(1,1) (1,1)> ?
                                  ├── 1
                                  └── 2
```

The sharing scopes are opened correctly but close immediately without including the choices. To understand how the empty scopes are created, we have a look at a simplified example where `head` occurs only once. For further simplification, the `share` implementation without IDs is used.

```
  share (cons coin nil) >>= headM
= (return $ do begin; x <- (cons coin nil); end; return x) >>= headM
= headM $ do begin; x <- (cons coin nil); end; return x
= headM $ do begin; end; return (Cons coin nil)
= do begin; end; headM (return (Cons coin nil))
= do begin; end; coin
```

Since `x <- (cons coin nil)` is inside the scope but `return x` is outside, this step moves the choice contained inside the list out of the sharing scope. For the previous example of adding the list's head twice, the last line would end with `addM coin coin`, which explains why no sharing is present. What is missing here is deep sharing, that is, sharing of the individual components of the list, so that the decision of the coins are linked.

Deep sharing is realized similar to the explicit-sharing library[3] which implements the

[3] http://hackage.haskell.org/package/explicit-sharing-0.9

approach of [Fischer et al., 2009] presented in subsection 2.4.2. At its core, deep sharing is implemented via a type class `Shareable` that all data types with shareable components need to implement.

```
class Shareable m a where
  shareArgs :: Monad n
            => (forall b. Shareable m b => m b -> n (m b))
            -> a -> n a
```

Although `shareArgs` is parameterized over a function that generalizes the type of the sharing operator, it is used only with `share` in this model. Similar to `share`, the function `shareArgs` adds a monad layer to its input.

TODO: Why?

When it comes to instances for types like lifted lists, the implementation is straightforward. The empty list does not need to be shared. For non-empty lists, the function `f`, that is, `share`, is applied to the components. Then the result is retrieved using bind and a list is constructed again. The additional monad layer required by the function type is implemented using `cons`, since the function is a smart constructor for a program that returns a list.

```
instance (Shareable m a) => Shareable m (List m a) where
    shareArgs f Nil = nil
    shareArgs f (Cons mx mxs) = do mz  <- f mx
                                   mzs <- f mxs
                                   cons mz mzs
```

With the implementation of deep sharing by means of `shareArgs`, we can finally define a sharing operator that covers nested choices, repeated sharing, nested sharing and deep sharing.

Nested choices, that is, multiple choices within one sharing scope, is implemented as part of the handler, which is discussed in the next section.

Repeated sharing required adding an ID and state code to distinguish different scopes. IDs are also required to link duplicated sharing scopes, so that they behave identically. Nested sharing required adding a `put` statement to the sharing scope, so that nested calls of `share` have a defined state to work with. In addition, it became clear that the namespace that supplies IDs for nested sharing needs to be distinct from the supply for repeated sharing, since the same IDs can otherwise be assigned unintentionally.

Lastly, we added deep sharing by defining type classes for normalization and sharing of components. The former moves effects from inside a complex data type to the root of the expression, so that handlers do not need to consider complex data types themselves. The latter defines a function `shareArgs` that allows us to not only share whole terms but also the individual components. The implementation of `share` now looks as follows.

```
instance (Share <: sig, State (Int, Int) <: sig, ND <: sig)
          => Sharing (Prog sig) where
  share p = do
    (i, j) <- get
    put (i + 1, j)
```

```
    return $ do
      begin (i,j)
      put (i, j + 1)
      x <- p
      x' <- shareArgs share x
      end (i,j)
      return x'
```

The ID supply is implemented using a state with two components which are incremented depending on the program layer. The outer layer, which is responsible for repeated sharing, increments the first component, while the second component is incremented in the inner program layer, which affects nested sharing.

We can observe the effect of `shareArgs` in the same example as before.

```
  share (cons coin nil) >>= headM
= (return $ do begin; x <- (cons coin nil);
    x' <- shareArgs share x; end; return x') >>= headM
= headM $ do begin;
    x' <- shareArgs share (Cons coin nil); end; return x'
= headM $ do begin;
    x' <- (share coin >>= \mz ->
           share nil  >>= \mzs -> cons mz mzs); end; return x'
= headM $ do begin;
    x' <- (share coin >>= \mz -> cons mz nil); end; return x'
= headM $ do begin; end; (share coin >>= \mz -> cons mz nil)
= do begin; end; (share coin >>= \mz -> headM (cons mz nil))
= do begin; end; share coin >>= id
```

The result still shows an empty scope from sharing the whole list. There is a difference, however, in the last part of the expression. Whereas `share` without deep sharing resulted in a simple coin, adding `shareArgs` wraps the coin in another call of `share`. Thus, the choice is shared correctly and the `share` operator behaves as expected.

3.4.5 Sharing Handler

The previous sections were focused on defining a program that models the different aspects of sharing syntactically, that is, scopes with the correct IDs should appear at the correct positions. What happens inside those scopes has not been discussed in detail, yet. Hence, this section focuses on handling the sharing effect.

```
runShare :: (ND :<: sig) => Prog (Share + sig) a -> Prog sig a
runShare (Return a)   = return a
runShare (BShare i p) = nameChoices [trip i 0] p
runShare (EShare _ p) = error "runShare: mismatched EShare"
runShare (Other op)   = Op (fmap runShare op)
```

Beginning with the top-level handler, there is not much of a difference to the first implementation of the handler in subsection 3.3.3. Although the structure is the same, the function `nameChoices` that handles the program inside the scope now has an additional argument of type `[Scope]`. A scope is a triple of integers where the first two digits represent the ID of a scope and the last digit is a counter. The function `trip` is a smart constructor for constructing triples from an ID and an initial counter value.

When a program inside a scope should be handled, the function `nameChoices` takes over. The signature is the same as for `runShare` except for a list of scopes. This list represents the the current scope environment, that is, how many scopes surround the current program. The third component of a scope becomes important when a choice is encountered.

```
nameChoices :: (ND :<: sig)
            => [Scope] -> Prog (Share + sig) a -> Prog sig a
nameChoices [] _ = error "nameChoices: missing scope"
nameChoices scopes@(i@(l,r,next):scps) prog =
  case prog of
    Return a      -> return a
    BShare i p    -> nameChoices (trip i 0 : scopes) p
    EShare i p    -> checkScope i scopes p
    Fail          -> fail
    Choice _ p q -> let f = nameChoices (inc i : scps)
                    in choiceID (Just i) (f p) (f q)
    Other op      -> Op (fmap (nameChoices scopes) op)
```

The ID of a scope is not enough to assign an ID to a choice because multiple choices can occur within the same scope. Thus, each scope has a counter that is incremented with the function `inc` when a choice has been assigned an ID, so that the next choice will receive an different ID. When a scope inside a scope is found, `nameChoices` continues handling the program but the ID the of the scope is added to the environment. Ending a scope is performed by the function `checkScopes` that is passed the ID of the ending tag, the scope environment and the remaining program.

```
checkScope :: (ND :<: sig)
           => SID -> [Scope] -> Prog (Share + sig) a -> Prog sig a
checkScope i scopes p =
  case scopes of
    []             -> error "checkScope: mismatched EShare"
    [(l,r,_)]      -> if (l,r) == i
                        then runShare p
                        else error "checkScope: wrong scope"
    ((l,r,_):scps) -> if (l,r) == i
                        then nameChoices scps p
                        else error "checkScope: crossing scopes"
```

There are three cases to distinguish when ending a scope. Firstly, the scope environment could be empty, that is, no scope has been opened. Since closing tags are supposed to follow opening tags, this is an error.

Secondly, the scope environment can have only one surrounding scope. In this case, the ID of the ending tag is checked against the current scope from the environment. If it matches, we leave the scope and let `runShare` handle the remaining program. If the tags do not match, this is an error.

Thirdly, the environment can contain more than one open scopes. Similar to the previous case, the tag IDs are compared. This time, matching tags means that we are still inside a scope. The current scope is left by removing the head element of `scopes` and `nameChoices` handles the remaining program. Here it becomes clear why we need to memorize a counter for each scope: When a nested scope interrupts handling the current scope, we must not begin counting from some initial value again after the nested scope is handled. Otherwise, the first choice of the scope and the first choice after the nested scope would receive the same ID.

3.5 Examples

As an example of a more complex program that makes use of explicit sharing, the implementation of permutation sort shown in subsection 3.3.3 is adapted. Conveniently, the second implementation is parameterized over instances of `MonadPlus` and uses the same lifted data types that are used to implement deep sharing. Thus, we only need to add explicit sharing to the `sort` function.

```
sort :: (MonadPlus m, Sharing m) => m (List m Int) -> m (List m Int)
sort l = do
  xs <- share (perm l)
  b <- isSorted xs
  guard b
  xs
```

Since the function is already generalized over instances of `MonadPlus`, explicit sharing is added as a type constraint by means of the class `Sharing`. The only function defined by `Sharing` is the sharing operator `share`, which depends on an instance of `Shareable`, that is, the function `shareArgs` for deep sharing must be defined.

```
class MonadPlus s => Sharing (s :: * -> *) where
    share :: Shareable s a => s a -> s (s a)
```

When comparing the runtime of the explicit sharing implementation with the naive, strict approach, there is considerable overhead generated by the lifted data type and effect handlers. Compared to 0.69 seconds for sorting a list of nine elements using the list monad, the same list now takes over three minutes to sort.

```
λ> testSort [7,6..1]
[1,2,3,4,5,6,7]
(2.56 secs)
λ> testSort [8,7..1]
[1,2,3,4,5,6,7,8]
(20.01 secs)
λ> testSort [9,8..1]
[1,2,3,4,5,6,7,8,9]
```

```
(196.08 secs)
```

However, there is one case where the explicit-sharing implementation is much faster. When the `isSorted` predicate is replaced by a constant function that returns `False`, lists of any length are "sorted" in constant time, while the list monad implementation still generates all permutations of the input list. Thus, the explicit-sharing implementation models non-strictness correctly. Since the resulting lists are indeed sorted, the issue of incoherent sharing mentioned in subsection 2.4.2 does not affect this implementation, either. The higher-order implementation is faster than the implementation with explicit scoping tags, presumably because less pattern matching is occurring due to the single sharing constructor, but sorting a list of nine elements still takes just over two minutes.

As another example, the idea of using sorting algorithms and a non-deterministic predicate to generate permutations introduced by Christiansen et al. [2016] is implemented for Quicksort. In order to incorporate sharing into the implementation, we use the function `partitionM` that splits a list into a pair of lists, depending on whether the elements fulfill a predicate or not. The result is shared and then accessed via lifted versions of the functions `fst` and `snd`.

```
quicksortM :: (Sharing m, MonadPlus m)
           => (m Int -> m Int -> m Bool)
           -> m (List m Int) -> m (List m Int)
quicksortM mp mxs = mxs >>=
  \xs -> case xs of
           Nil -> nil
           Cons my mys ->
             do p <- share (partitionM (mp my) mys)
                appM (appM (quicksortM mp (first p)) (cons my nil))
                     (quicksortM mp (second p))
```

When `quicksortM` is called with the non-deterministic predicate `\_ _ -> coin`, the function returns all $n!$ permutations for a list of length n. The Quicksort algorithm is interesting because even a small input list like `[1..4]` generates many sharing scopes. When observing the IDs of scopes that are generated, up to ten levels of nested and repeated sharing occur for a list of four elements. This leads to one reason for the lacking performance of the implementation, which can be observed by adding following line to the handler of the sharing effect.

```
runShare (BShare _ (EShare _ p)) = trace "empty scope" (runShare p)
```

While computing the permutations of a four-element list, the output `empty scope` appears 32,010 times, while only 111 choices are assigned an ID. One big issue of the implementation is the creation of empty sharing scopes by `shareArgs`. Since deep sharing is not demand-driven, a lot of sharing scopes are created preemptively. In the example, this accounts for roughly 25,000 empty scopes. The remaining empty scopes can be attributed to the monadic structure that sometimes moves the argument of `share` outside of the sharing scope, as seen in the previous section.

Nevertheless, instead of a highly optimized implementation, the goal of this thesis is to model Curry's call-time choice correctly in both Haskell and Coq for the purpose of reasoning about programs. In the following chapter, the current model is implemented in Coq.

4 Call-Time Choice modelled in Coq

The goal of this chapter is to transfer one of the Haskell implementation approaches of the call-time choice model to Coq. Before implementing a specific approach, the obstacles that Coq's more restrictive language entails are discussed. We begin with the data structure `Prog`, that is, the free monad, which allows us to model programs with effects of type `sig` and results of type `a`. In Haskell, the data type looks as follows.

```
data Prog sig a = Return a | Op (sig (Prog sig a))
```

The direct translation of the definition to Coq looks very similar to the Haskell version, aside from renaming and the explicit constructor types.

```
Fail Inductive Free F A :=
| pure : A -> Free F A
| impure : F (Free F A) -> Free F A.
```

However, the definition is rejected by Coq upon loading the file with the following error message.

```
Non-strictly positive occurrence of "Free" in "F (Free F A) -> Free F A".
```

Non-strictly positive occurrence is the main challenge when modelling effects in Coq. The reason for this error is explained in the next section.

4.1 Non-strictly Positive Occurrence

- What does non-strictly positive occurrence mean?
- Motivation for usage of containers

In section 2.1, we learned that Coq distinguishes between non-recursive definitions and functions that use recursion. The reason for this is that Coq checks functions for termination, which is an important part of Coq's proof logic. To understand why functions must always terminate in Coq, we consider the following function.

```
Fail Fixpoint loop (x : unit) : A := loop x.
```

The function receives an argument `x` and calls itself with the same argument. Since this function obviously never terminates, the result type `A` is arbitrary. In particular, we could instantiate `A` with `False`, the false proposition. The value `loop tt : False` could be used to prove anything, according to the principle of explosion. For this reason, Coq requires all recursive functions to terminate provably.

Returning to the original data type, what is link between `Free` and termination? It is well known that recursion can be implemented in languages without explicit recursion syntax by means of constructs like the Y combinator or the data type `Mu` for type-level recursion.

```
Fail Inductive Mu A := mu : (Mu A -> A) -> Mu A.
```

`Mu` is not accepted by Coq for the same reason as `Free`: non-strictly positive occurrence of the respective data type. The problematic property of non-strictly positive data types is that the type occurs on the left-hand side of a constructor argument's function type. This would allow general recursion and thus, as described above, make Coq's logic inconsistent.

In case of `Free`, the non-strictly positive occurrence is not as apparent as before because the constructors do not have functional arguments. However, `F` is being applied to `Free F A`. If `F` has a functional argument with appropriate types, the resulting type becomes non-strictly positive, as shown below.

```
Definition Cont R A := (A -> R) -> R.

(* Free (Cont R) *)
Fail Inductive ContF R A :=
| pureC   : A -> ContF R A
| impureC : ((ContF R A -> R) -> R) -> ContF R A.
```

In the type of `impureC` contains a non-strictly positive occurrence of `ContF R A`. Consequently, Coq rejects `Free` because it is not guaranteed that no instance violates the strict positivity requirement. Representing the `Free` data type therefore requires a way to restrict the definition to strictly positive data types. One approach to achieve this goal is described in the next section.

4.2 Containers

- How do containers work?
- How do we translate effect functors into containers?

Containers, as introduced by Abbott et al. [2005], are an abstraction of data types that store values, with the property that only strictly positive functors. can be modelled as a container. This will allow us to define a version of `Free` that works with containers of type constructors instead of the type constructors itself. In the following, we have a more detailed look at containers and how they relate to functors.

4.2.1 First-Order Containers

For a functor `F`, we define a first version of a container class that looks as follows.

```
Class Container F :=
  {
    Shape   : Type;
    Pos     : Shape -> Type;
  }.
```

The first component of a container is the type `Shape`. A shape determines how the data type is structured, regardless of the stored values. For example, the shape of a list is the same as the shape of Peano numbers: a number that represents the length of the list, or rather the number of `Cons`/`Succ` applications. A pair, on the other hand, has only a single shape.

The second component of a container is a function `Pos : Shape -> Type` that gives each shape a type that represents the positions within the shape. In the example of pairs, the shape has two positions, the first and second component. Each element of a list is a position within the shape. Therefore, the position type for lists with length n is natural numbers smaller than n. Peano numbers do not have elements and therefore, the position type for each shape is empty.

Containers can be extended by a function that maps all valid positions to values. Since the position type depends on a concrete shape, the definition in Coq is quantified universally over values of type `Shape`. In the following definition, `Shape` and `Pos` represent the components of a container as defined above.

```
Inductive Ext Shape Pos A :=
| ext : forall s, (Pos s -> A) -> Ext Shape Pos A.
```

The extension of a container is isomorphic to the original data type. This means that we can define functions `to` and `from` that transform values of the container extension into values of the original data type and vice versa. Additionally, proofs that show `to . from = id` and `from . to = id` as evidence of the isomorphism are required. Thus, the `Container` class is extended by the transformation functions and the isomorphism proofs.

```
Class Container F :=
  {
    Shape   : Type;
    Pos     : Shape -> Type;
    to      : forall A, Ext Shape Pos A -> F A;
    from    : forall A, F A -> Ext Shape Pos A;
    to_from : forall A (fx : F A), to (from fx) = fx;
    from_to : forall A (e : Ext Shape Pos A), from (to e) = e
  }.
```

To gain a better understanding of how functors are represented using containers, the following subsection describes the general scheme for translating a functor into an isomorphic container[1].

[1]Often the container extension is meant when talking about containers. As described in the container class, only the extension of a container is isomorphic to the original data type but not the container alone.

4.2.2 Modelling Functors as Containers

As an example of modelling a functor as a container, we consider the non-determinism effect described in subsection 3.2.3 The definition of the data type is simply translated from Haskell.

```
Inductive Choice (A : Type) :=
| cfail   : Choice A
| cchoice : option ID -> A -> A -> Choice A.
```

When determining the shape of a functor, we first have to consider whether the data type is recursive. For the effect data types, recursion is introduced by `Free`, so the types are generally non-recursive. This means that we only need one shape for each constructor.

Next, we focus on the arguments of the constructors. Since `cfail` has no arguments, there is no data that needs to be stored in its shape. `cchoice`, however, has three arguments. In the definition of the container extensions in the previous subsection, a function of type `forall s, Pos s -> A` was mentioned. This function is responsible for all arguments of type `A` and thus, they are not part of the shape of the `cchoice` contructor. All arguments of types other than `A`, that is, `option ID` in this case, become part of the shape. This results in the following shape for the type `Choice`.

```
Inductive Shape__Choice :=
| sfail : Shape__Choice
| schoice : option ID -> Shape__Choice.
```

The second component of a container is the function `Pos` that assigns each shape a position type. This type describes all positions for values of type `A` in the constructor that the shape represents. For the `cfail` constructor, there are no arguments (of type `A`). This means that when `Pos` receives a shape `sfail`, a type that represents no positions needs to be returned. Thus, the constructorless type `Inductive Void :=.` is returned. On the other hand, `cchoice` contains two arguments of type `A` and thus, has two positions. We could define a new type with two constructors, but for this simple task, `bool` works just as well. Consequently, `true` means the first `A` position in `cchoice` and `false` the second one.

```
Definition Pos__Choice (s: Shape__Choice) : Type :=
  match s with
  | sfail     => Void
  | schoice _ => bool
  end.
```

With the definition of `Shape` and `Pos`, we can define the container extension type as follows.

```
Definition Ext__Choice : Type -> Type := Ext Shape__Choice Pos__Choice.
```

The transformation functions are a first indicator whether the definition of the shape and position types are correct. When transforming a value of type `Choice`, we need to supply two arguments to the container extension constructor `ext`. Firstly, the shape that corresponds to the constructor. For `cfail` and `cchoice`, this is `sfail` and `schoice` respectively. The latter takes the optional ID argument since this information cannot be stored in the second argument `ext`: the position function. To construct such a function, we define a function `pf` that takes a position argument. The type of the position is determined by `Pos` applied to the first argument of `ext`, that is, the shape that corresponds to the constructor that is currently handled. Evaluating `Pos__Choice sfail` yields `Void`, so we need to return a value of type `A` for each constructor of the type. Since `Void` does not have constructors, we do not need to return anything. This is expressed in Coq by doing empty pattern matching on the position argument of the function. The type of the position is known at compile time and thus, Coq accepts this definition.

```
Fixpoint from__Choice A (z : Choice A) : Ext__Choice A :=
  match z with
  | cfail _     =>
    let pf (p : Pos__Choice sfail) := match p with end
    in ext sfail pf
  | cchoice mid l r =>
    let s := schoice mid
    in let pf (p : Pos__Choice s) := if p then l else r
       in ext s pf
  end.
```

The position function in case of `cchoice` is slightly more involved. This time, `Pos__Choice` returns `bool` as the type of the argument `p`. Consequently, we need to return a value of type `A` for both members of the type `bool`. Since `cchoice` has two such arguments, we just need to return the corresponding argument depending on whether `p` is `true` or `false`.

Based on this intuition of how the position function works, `to` can be defined easily. We pattern match for the different shapes and create values using the corresponding constructors of the original data type. This time, there is a position function `pf` given by the container extension. Knowing the type of argument it expects, we use the function to retrieve the values of type `A`.

```
 Definition to__Choice A (e: Ext__Choice A) : Choice A :=
   match e with
   | ext sfail         pf => cfail A
   | ext (schoice mid) pf => cchoice mid (pf true) (pf false)
   end.
```

Finally, the remaining parts of the container definition are the isomorphism proofs. The first proof is done by case distinction over the value `ox`.

```
Lemma to_from__Choice : forall A (ox : Choice A),
                          to__Choice (from__Choice ox) = ox.
  Proof.
    intros A ox.
    destruct ox; reflexivity.
```

```
Qed.
```

In both cases, the transformation functions neither add nor remove information, as shown in the following for the cchoice constructor.

```
  to__Choice (from__Choice (cchoice mid l r))
= let s := schoice mid
  in let pf (p : Pos__Choice s) := if p then l else r
     in to__Choice (ext s pf)
= let s := schoice mid
  in let pf (p : Pos__Choice s) := if p then l else r
     in cchoice mid (pf true) (pf false)
= cchoice mid l r
```

The second proof is slightly more complicated than the first one. Whereas we had to show the equality of two values of type `Choice` before, we now need to do the same for values of type `Ext__Choice`. Since such values contain function arguments, we need to show function equality.

```
Lemma from_to__Choice : forall A (e : Ext__Choice A),
  from__Choice (to__Choice e) = e.
Proof.
  intros A [s pf].
  destruct s; simpl; f_equal; extensionality p.
  - contradiction.
  - destruct p; reflexivity.
Qed.
```

After destructing the shape of `e` for a case distinction and simplifying the expression, we are in a situation where the goal is similar to `ext sfail pf = ext sfail pf'`. The tactic `f_equal` states that two values are equal if both constructors are the same and all arguments are equal, too. Since the constructor and first argument are equal, the only remaining goal is to prove that the position functions are equal. The tactic `extensionality` is useful to show function equality, that is, $\forall x : f x = g x \implies f = g$. The position `p` appears in the proof context with an appropriate type, which is determined by the function `Pos__Choice`. There are two goals left to prove that correspond to the shapes of the container. The first goal arises from `sfail` and looks as follows.

```
A : Type
pf : Void -> A
p : Void
============================
match p return A with
end = pf p
```

Since the position function for `sfail` takes arguments of type `Void`, there is a value `p` of type `Void` in the context. However, this is not possible because `Void` has no constructors. Consequently, the tactic `contradiction` solves this case.

For the second case that corresponds to the shape `schoice`, the position function takes arguments of type `bool`.

```
A : Type
o : option ID
pf : bool -> A
p : bool
============================
(if p then pf true else pf false) = pf p
```

If we destruct `p`, we can see that the `if`-statement returns `pf true` if `p` is `true` and vice versa, which leads to two trivially true equations. This shows that the transformation functions form an isomorphism. Therefore, strictly positive functors can be represented as container extensions.

Finally, the container instance can be defined as follows.

```
Instance C__Choice : Container Choice :=
  {
    to_from := to_from__Choice;
    from_to := from_to__Choice
  }.
```

The isomorphism contains all necessary information about the container and its extension. Thus, Coq is able to infer the omitted definitions.

4.3 Modelling Effects

- In which ways is the Coq implementation simplified, compared to Haskell?
- How does the adapted Prog/sig infrastructure work?
- How do we translate recursive functions?

In the previous section, a technique for representing functors as containers was presented. The initial motivation for this was the issue of non-strictly positive occurrence errors in the definition of `Free`. Since containers can only represent strictly positive functors, the following definition is accepted by Coq.

```
Variable F : Type -> Type.

Inductive Free (C__F : Container F) A :=
| pure   : A -> Free C__F A
| impure : Ext Shape Pos (Free C__F A) -> Free C__F A.
```

The parameter `F` is replaced by the container that represents `F`. Instead of writing `F (Free F A)` as the first argument of `impure`, we can now use the extension of the container `C__F` applied to `Free C__F A`.

4.3.1 Infrastructure

With the `Free` data type defined, we can begin implementing the remaining infrastructure. In the Haskell implementation, various type classes and language extensions were used to make working with effects as comfortable as possible. For the Coq implementation, the infrastructure is not implemented as generalized as before, but rather tailored to the purpose of modelling call-time choice. This means, for example, that handlers do not work with arbitrary signatures where one part is a type variable.

Combining Effects Although it would be possible to merge all effects into one call-time choice effect with a large handler, the combination of independent effects is still a good idea in terms of modularity. Thus, we need a counterpart to the functor `:+:` in Haskell.

The original functor is still parameterized over two functors `F` and `G` that represent effects.

```
Variable F G : Type -> Type.

Inductive Comb A : Type :=
| Inl : F A -> Comb A
| Inr : G A -> Comb A.
```

Since `Comb` combines effects into a new effect functor, we need to represent the combination as a container, too. We assume that we have containers `C__F : Container F` and `C__G : Container G` for both functors. The shape can then be defined as a `sum` type – the Haskell equivalent is `Either` – of the containers' shapes.

```
Definition Shape__Comb : Type := sum (@Shape F C__F) (@Shape G C__G).
```

The same principle applies to the function `Pos`. The shape argument can be distinguished by means of the `sum` constructors `inl` and `inr`. Depending on the `sum` constructor, the shape inside is passed to the `Pos` function of either `C__F` or `C__G`.

```
Definition Pos__Comb (s : Shape__Comb) : Type :=
  match s with
  | inl x => @Pos F C__F x
  | inr x => @Pos G C__G x
  end.
```

The transformation functions also depend on the functions defined in the containers for `F` and `G`. To transform an `Ext__Comb` into a `Comb`, the `sum` constructor that wraps around the shape is removed and the original `to` function is called. The result is then wrapped in a `Inl` or `Inr` constructor. Transforming a `Comb` into an `Ext__Comb` is achieved by passing the value inside the `Comb` constructor to `from` and then wrapping a `sum` constructor around the shape of the result.

The isomorphism proofs work similar to the `Choice` proofs in regard to destructing values and shapes. In addition, the isomorphism properties arising from $C_F \cong F$ and $C_G \cong G$ are applied.

4.4 Sharing

- Laws of sharing

4.5 Extra

4.5.1 Higher-Order Containers

In subsection 3.3.2, the higher-order approach of modelling effect signatures not as a functor, but as as higher-order functor of type `(* -> *) -> * -> *` was discussed. Similar to the first-order data type for `Prog`, the following higher-order version fails due to non-strictly positive occurrence of `Free`.

```
Fail Inductive Free F A :=
| pure : A -> Free F A
| impure : F (Free F) A -> Free F A.
```

This calls for a higher-order container that is able to represent higher-order functors. Initially, one could assume that we can adapt the first-order container by replacing the functor `F` with the higher-order functor `H` and adding the necessary functor arguments of `H` to the list of quantified variables, as follows.

```
Class HContainer (H : (Type -> Type) -> Type -> Type) :=
  {
    Shape   : Type;
    Pos     : Shape -> Type;
    to      : forall F A, Ext Shape Pos F A -> H F A;
    from    : forall F A, H F A -> Ext Shape Pos F A;
    to_from : forall F A (fx : H F A), to (from fx) = fx;
    from_to : forall F A (e : Ext Shape Pos F A), from (to e) = e
  }.
```

The extension of such a higher-order container needs to be adapted, too. Reconsidering higher-order functor data types, such as choice, the type variables `m` and `a`

```
Inductive Ext Shape (Pos : Shape -> Type) (F : Type -> Type) A :=
  ext : forall s, (Pos s -> F A) -> Ext Shape Pos F A.
```

5 Curry Programs modelled in Coq

- Can we use the Coq model of call-time choice to prove properties about actual Curry programs?

6 Conclusion

```
  Class Container_Asd F :=
  {
    Shape : Type;
    Pos : Shape -> Type;
    to : forall A, Ext Shape Pos A -> F A;
    from : forall A, F A -> Ext Shape Pos A;
    to_from : forall A (fx : F A), to (from fx) = fx;
    from_to : forall A (e : Ext Shape Pos A), from (to e) = e
  }.

  Definition join (M: Type → Type) `(Monad M) A (mmx : M (M A)) : M A := bind _

End MonadClass.
Arguments join {_} {_} {_}.

Section MonadInstance.

  Variable F : Type → Type.
  Variable C__F : Container F.
```

Bibliography

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. Theor. Comput. Sci., 342(1):3–27, September 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2005.06.002. URL http://dx.doi.org/10.1016/j.tcs.2005.06.002.

Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. KiCS2: A new compiler from curry to haskell. In Proceedings of the 20th International Conference on Functional and Constraint Logic Programming, WFLP'11, pages 1–18, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22530-7. URL http://dl.acm.org/citation.cfm?id=2032603.2032605.

Jan Christiansen, Nikita Danilenko, and Sandra Dylus. All sorts of permutations (functional pearl). SIGPLAN Not., 51(9):168–179, September 2016. ISSN 0362-1340. doi: 10.1145/3022670.2951949. URL http://doi.acm.org/10.1145/3022670.2951949.

Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely functional lazy non-deterministic programming. SIGPLAN Not., 44(9):11–22, August 2009. ISSN 0362-1340. doi: 10.1145/1631687.1596556. URL http://doi.acm.org/10.1145/1631687.1596556.

Wouter Swierstra. Data types à la carte. J. Funct. Program., 18(4):423–436, July 2008. ISSN 0956-7968. doi: 10.1017/S0956796808006758. URL http://dx.doi.org/10.1017/S0956796808006758.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. ACM SIGPLAN Notices, 49, 09 2014. doi: 10.1145/2633357.2633358.