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

February 5, 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].

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.

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.

```
*Test> sort [9, 8..1] :: [[Int]]
[[1,2,3,4,5,6,7,8,9]]
(0.69 secs, 717,914,784 bytes)
*Test> sort [10, 9..1] :: [[Int]]
[[1,2,3,4,5,6,7,8,9,10]]
(6.67 secs, 7,437,960,280 bytes)
*Test> sort [11, 10..1] :: [[Int]]
[[1,2,3,4,5,6,7,8,9,10,11]]
(77.54 secs, 84,743,416,080 bytes)
```

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 as the smart constructors and a lifted `head` function as follows.

```
> 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 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 Choice' 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 `Choice` does not have values as arguments but rather programs that return values.

```
data ChoiceProg a = Return a
                  | Op (Choice' (ChoiceProg a))
```

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

```
data ChoiceProg a = Return a
                  | OpFail
                  | OpChoice (ChoiceProg a) (ChoiceProg 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, Applicative (Free 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

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

3.3 Sharing

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

4 Call-Time Choice modelled in Coq

The goal of this chapter is to transfer the Haskell implementation of call-time choice to Coq. We begin with the data structure `Prog`, that is, the free monad, which allowed us to model programs with effects of type `sig` and results of type `a`.

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

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

```
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".
```

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 are an abstraction of data types that store values, with the property that only strictly positive data types 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. First, however, we will have a more detailed look at containers.

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`.

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

The extension of a container models the concrete data type.

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?

4.4 Sharing

- Laws of sharing

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

Bibliography

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`.

Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely functional lazy nondeterministic 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`.

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