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

January 29, 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

2.2 Haskell

2.2.1 Monad and MonadPlus

2.3 Curry

2.3.1 Non-strictness

2.3.2 Sharing

2.3.3 Non-determinism

2.4 Modelling Curry Programs using Monadic Code Transformation

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
```

```
*Test> sort [8,7..0] :: [[Int]]
[[0,1,2,3,4,5,6,7,8]]
(1.05 secs, 717,910,424 bytes)
*Test> sort [9,8..0] :: [[Int]]
[[0,1,2,3,4,5,6,7,8,9]]
(9.64 secs, 7,437,956,760 bytes)
*Test> sort [10,9..0] :: [[Int]]
[[0,1,2,3,4,5,6,7,8,9,10]]
(107.09 secs, 84,743,412,160 bytes)
```

As mentioned before, modelling non-determinism in Haskell requires us to adapt data types so that components can contain non-determinism because otherwise non-strictness is lost.

TODO: Copy examples? New examples? No examples?

```
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.

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, 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. 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. This will be explained in more detail later.

3 Call-Time Choice modelled in Haskell

3.1 Free Monads

3.2 Modelling Effects

3.3 Sharing

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

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

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.

5 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`.