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 3, 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

Contents

1 Introduction

2 Preliminaries

2.1 Coq

2.2 Haskell

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.

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

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`.